Edgar H. Sibley
Panel Editor

DARTS—a design method for real-time systems—leads to a highly structured modular system with well-defined interfaces and reduced coupling between tasks.

A SOFTWARE DESIGN METHOD FOR REAL-TIME SYSTEMS

H. GOMAA

Real-time systems typically consist of several concurrent processes or tasks. Each task is sequential, and concurrency is obtained by having asynchronous tasks running at different speeds. From time to time, the tasks need to communicate and synchronize with each other.

This paper surveys briefly several existing software design methods and considers how suitable they are for real-time systems. It outlines the requirements of real-time systems design and describes a real-time design method known as DARTS (Design Approach for Real-Time Systems). An example, the design of a robot controller, is given to illustrate the use of the DARTS method.

SURVEY OF SOFTWARE DESIGN METHODS

The software design methods surveyed in this section are the Jackson and Warnier methods, Structured Design, and the Higher Order Software, Mascot, and Information Hiding methods. A more detailed review is given in [6].

The Jackson [9] (also known as Jackson Structured Programming) and Warnier [11] design methods are data-structure-oriented design methods primarily applicable to program design. Neither method handles the problem of decomposing a system into modules or tasks, and consequently neither is appropriate for real-time systems design.

Structured Design [10, 15] is a systems design method that deals with decomposing a system into modules. It can lead to highly functional modular designs but provides no help with structuring a system into tasks. Nor does it address the issue of internal module design. Structured Design is often used in conjunction with Structured Analysis [3, 5], which makes use of data flow diagrams and functional decomposition.

The Higher Order Software design method [8] also uses functional decomposition for decomposing a system into modules, but it, too, fails to address the issue of decomposing a system into concurrent tasks.

The Mascot method [14] is well suited for real-time systems since it deals specifically with structuring a system into tasks and defining the interfaces between them. However, it starts with a network diagram of tasks and addresses neither the issue of *how* to structure a system into tasks nor the structure of the individual tasks themselves.

Parnas' Information Hiding method [12] is a powerful design concept that leads to highly modular systems with low module coupling. The technique has been adopted by other design methods such as Structured Design, Mascot, and Object Oriented Design [1].

REQUIREMENTS OF A REAL-TIME SYSTEMS DESIGN METHOD

The above survey of design methods shows that each method has some limitations in terms of real-time systems design. The two methods that come closest to satisfying the needs of real-time systems design are Structured Design and Mascot. This section identifies the essential requirements of an adequate real-time systems design method and points out to what degree these requirements are satisfied in existing design methods.

Data-Flow-Oriented Design

The data flow approach to software design is particularly appropriate for the design of real-time systems because the data in these systems may be considered to flow from input to output and in between to be transformed by software tasks.

The data-flow-oriented methods are best exemplified by Structured Analysis and Structured Design, which are frequently used together. With Structured Analysis, data flow diagrams are used to show the functions (transforms) of a system as well as the data flows between the transforms and the data stores accessed by them. Other features are the use of a data dictionary and the hierarchical decomposition of transforms.

Structured Design [15], also known as Composite Design [10], consists of two main components: (1) two sets of criteria, cohesion and coupling, which are used for evaluating the quality of a design; and (2) a design method for guiding designers in a top-down decomposition of a system into modules. The objective of Structured Design is to produce a design in which modules have high cohesion and low coupling.

The Structured Design method consists of two design approaches, Transform Centered Design and Transaction Centered Design. With Transform Centered Design, the major streams of data are identified as they flow and are transformed from external input to external output. The system is then structured so that each major abstract input stream, each major abstract output stream, and each major transformation has a corresponding branch in the structure chart. Transaction Centered Design is applicable where the data flow consists of data or control information that is passed to a transform initiating some action or sequence of actions based on the incoming data.

Because real-time systems are usually data flow oriented, the DARTS method starts with a data flow analysis of the system.

Task Communication and Synchronization

Because it is essential in real-time systems for tasks to communicate and synchronize their operations, most real-time operating systems support some mechanism for task communication and/or synchronization. The most common of these mechanisms are discussed below.

Task Synchronization. Two kinds of task synchronization found in real-time systems are mutual exclusion and cross stimulation. Mutual exclusion is typically required when shared data can be accessed concurrently by two or more tasks [4]. It is enforced by means of binary semaphores. Cross stimulation occurs when one task is awaiting a signal from another task before it can proceed. Binary semaphores and event synchronization can be used to effect cross stimulation. Both mutual exclusion and cross stimulation are used for task synchronization in DARTS.

Task Communication. Task communication occurs when a producer task needs to pass information to a consumer task. The most common form of task communication is Message Communication [2]. The communication may be closely coupled (i.e., each time the producer sends a message, it waits for a response from the consumer), or it may be loosely coupled (i.e., the producer and consumer proceed at their own rates and a queue of messages builds up between the producer and consumer). In either case, if the consumer requests a message from the producer and the queue is empty, the consumer has to wait until a message becomes available.

The message communication mechanism is provided in one of three ways: by the operating system; by providing multitasking with a task communication capability in the implementation language (e.g., Ada [1]); or by means of a module that handles message communication, using the synchronizing primitives provided by the operating system. This is the approach used in Mascot where tasks communicate with each other via channels. Channels are used for passing data, such as messages, between tasks.

Both loosely and closely coupled message communication are supported in DARTS.

Information Hiding

The concept of Information Hiding (also known as data abstraction) was introduced by Parnas [12] as a criterion for decomposing a system into modules. The objective is to hide key design decisions; that is, each key design decision should be known to only one module. With Information Hiding, information sharing between modules is kept to a minimum.

The advantage of this method is that modules are more self-contained and the system more modifiable and thus more maintainable. The disadvantage is the overhead consumed by accessing a data structure via a function rather than directly.

Mascot uses the Information Hiding concept to the extent that access to data in channels (which may be used for message communication) and pools (used for shared data) is provided only by means of access procedures. In this way, both the details of the data structure and the synchronization of access to the data structure are hidden from the calling task.

In DARTS, as in Mascot, Information Hiding is used to define task interfaces. Two classes of task interface modules are supported—Task Synchronization Modules (TSM) and Task Communication Modules (TCM), which minimize coupling between tasks.

State Dependency in Transaction Processing

Many real-time systems are transaction oriented or incorporate some degree of transaction processing (i.e., the action or sequence of actions to be carried out depends on the nature of the incoming data). Transaction

Centered Design, a component of the Structured Design method [15], addresses this issue although it has a serious limitation in that it does not deal with state dependency in transaction processing. DARTS overcomes this limitation.

DARTS

The DARTS design method starts with the Requirements Specification, which defines what features the system will provide with no consideration as to how they will be provided. As a given specification can be designed and implemented in many different ways, the development of data flow diagrams is considered the first phase of the design process. At this stage, the system is decomposed into subsystems and the subsystem interfaces identified.

The DARTS design method can be thought of as extending the Structured Analysis/Structured Design method by providing an approach for structuring the system into tasks as well as a mechanism for defining the interfaces between tasks. In this sense, it draws on the experience gained in concurrent processing. As with other design methods, DARTS is intended to be iterative. The steps in the DARTS design method are described below.

Data Flow Analysis

Data flow diagrams are used as an analysis tool. Starting with the functional requirements of the system, the data flow through the system is analyzed and the major functions determined. The data flow diagrams are developed and decomposed to sufficient depth to identify the major subsystems and the major components of each subsystem.

Each data flow diagram contains transform bubbles representing functions carried out by the system, arrows representing data flows between transforms, and data stores representing data repositories.

A data dictionary defines the data items contained in the data flows and data stores.

Decomposition into Tasks

Having identified all the functions in the system and the data flows between them, we are now in a position to identify concurrency. The next stage of the DARTS method therefore involves determining how concurrent tasks will be identified on the data flow diagram.

The main consideration in decomposing a software system into concurrent tasks is the asynchronous nature of the functions within the system. The transforms in the data flow diagrams are analyzed to identify which may run concurrently and which are sequential in nature. By this means, tasks are identified: One transform may correspond to one task, or one task may encompass several transforms.

The data flow diagrams are now redrawn showing the tasks and their interfaces. In so doing, a box is drawn around each transform or set of transforms that logically form a task. Each box then becomes a task.

The criteria for deciding whether a transform should be a separate task or grouped with other transforms into one task are the following.

Dependency on I/O. Depending on input or output, a transform is often constrained to run at a speed dictated by the speed of the I/O device with which it is interacting. In this case, the transform needs to be a separate task.

Time-critical functions. A time-critical function needs to run as a high priority and therefore needs to be a separate task.

Computational requirements. A computationally intensive function (or set of functions) can run as a lower priority task consuming spare CPU cycles.

Functional cohesion. Transforms that perform a set of closely related functions can be grouped together into a task. Since the data traffic between these functions may be high, having them as separate tasks will increase system overhead, whereas implementing each function as a separate module within the same task ensures functional cohesion both at the module and task levels.

Temporal cohesion. Certain transforms perform functions that are carried out at the same time. These functions may be grouped into a task so that they are executed each time the task receives a stimulus.

Although temporal cohesion is not considered a good module decomposition criterion in Structured Design, it is considered in DARTS to be acceptable at the task level. Each function should be implemented as a separate module to achieve functional cohesion at the module level. These modules in turn are grouped into the task thereby achieving temporal cohesion at the task level.

Periodic execution. A transform that needs to be executed periodically can be structured as a separate task that is activated at regular intervals.

When a system is structured into tasks, the tasks may all run on the same processor or may be split among two or more processors. The design decisions to be made at this stage are based on various factors such as system performance.

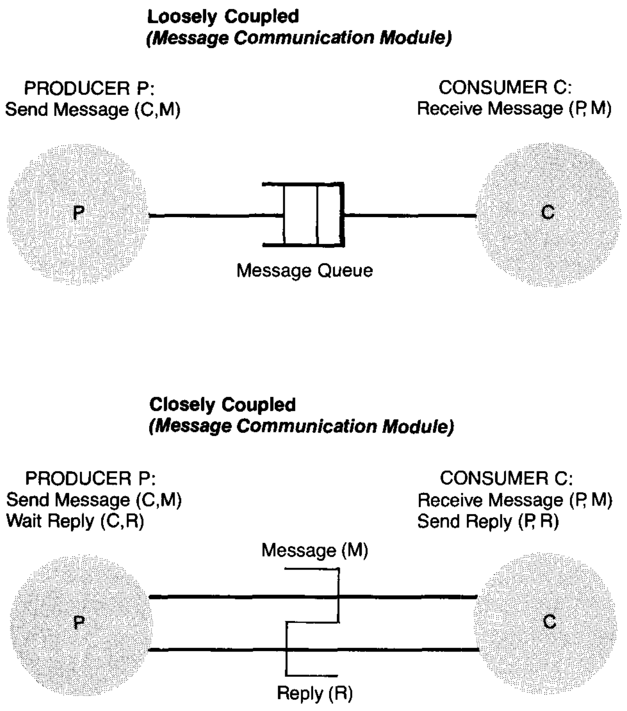
Definition of Task Interfaces

It is now time to consider the interfaces between tasks. On the data flow diagrams, the interfaces are in the form of data flows or data stores. The next stage involves formalizing the task interfaces.

In DARTS, task interfaces are handled by defining two classes of task interface modules, Task Communication Modules (TCM) and Task Synchronization Modules (TSM).

Task Communication Modules. A TCM handles all cases of communication among tasks. Typically, a TCM contains a data structure and defines the access procedures to it.

Conceptually, a TCM always runs in the task that invokes it. Thus, it is possible for a TCM to execute


Loosely Coupled
(Message Communication Module)

Closely Coupled
(Message Communication Module)

FIGURE 1. Message Communication

concurrently within two tasks. It is therefore essential that access procedures provide the synchronization and mutual exclusion conditions necessary to ensure consistent and correct access to the data.

A TCM makes use of the synchronizing primitives provided by the operating system. Thus, the implementation of a TCM will vary from system to system, but conceptually its function will be similar.

Two different types of TCMs are supported in DARTS:

Message Communication Module. Message communication is handled by a TCM called the Message Communication Module (MCM). MCMs support both loosely coupled and closely coupled message communication.

In loosely coupled message communication, the message queue includes binary semaphores for controlling mutual exclusion. Event synchronization is used for suspending the producer when the queue becomes full and suspending the consumer when the queue is empty. Access routines are provided for sending and receiving messages as well as getting and releasing message blocks. Furthermore, a maximum size is imposed on each message queue.

In the case of closely coupled message communication, the maximum size of the queue is reduced to one element. Sending and receiving of replies are supported by having a one-element message queue in each direction—one for messages and one for replies.

In addition, chiefly in loosely coupled communication, a task may wait for a message or reply to arrive at any one of several message queues. The task is activated when a message or reply arrives. This is achieved by having each message queue associated with an event occurrence. Adding a message to an empty queue results in an event's being signaled and the task activated.

The message communication mechanisms supported in DARTS and the graphical notation for loosely coupled and closely coupled message communication are shown in Figure 1.

Information Hiding Module. The concept of a pool or data store is required for data used for reference purposes. The shared data are accessible to two or more tasks either for read only or read/write purposes. A TCM called the Information Hiding Module (IHM) is used for this purpose. The IHM defines the data store as well as the access procedures to it.

Figure 2 shows the graphical notation used in DARTS for an IHM. The data store is shown as a box, and the access procedures are conceptually executed in tasks A and B. The arrows indicate the data flows between task and data store.

Task Synchronization Module. Events are used for synchronization purposes between tasks where no actual information transfer is needed. A destination task may wait for an event occurrence, or a source task may signal an event that activates the destination task. The graphical notation used for task synchronization in DARTS is given in Figure 3.

In DARTS, the synchronization mechanism is extended to allow one task to wait for any one of several events to be signaled. If any one event is signaled, the task is activated. A task may wait for events used only for synchronization purposes as well as events associated with message queues.

The primitives for signaling an event and waiting for an event are provided by the operating system. Waiting

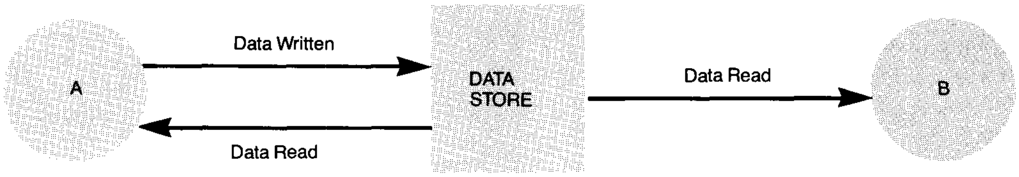


FIGURE 2. Information Hiding Module

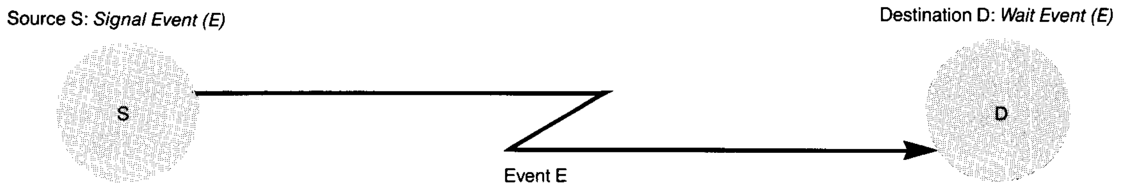


FIGURE 3. Task Synchronization

for a number of events can be a more complicated synchronization situation, and, for this reason, the concept of a TSM is introduced.

A TSM is typically the main or supervisory module of a task. There is usually only one TSM per task, and it is required only for tasks that do a significant amount of task synchronization. In this module, the task waits for one or more events to occur; these may be synchronizing events or message queuing events. Depending on the circumstances, the task may wait for different events at different times.

Task Interfaces. In DARTS, task interfaces are formalized according to the following guidelines.

A data flow between two tasks is treated as one of the following:

1. A loosely coupled message queue if one task needs to pass information to the other and the two tasks may proceed at different speeds. This message queue is handled by an MCM.
2. A closely coupled message/reply if information is passed from one task to another, but the first task cannot proceed until it has received a reply from the second. This is also handled by an MCM.
3. An event signal if only a notification of an event occurrence and no data transfer are required.

A data store that needs to be accessed by two or more tasks is handled as an IHM in which the data structure is defined as well as the access routines to the data structure.

In addition, each task that waits for one or more events may need a TSM.

Task Design

Structured Design. The next stage of the DARTS method involves designing each individual task, where each task represents a sequential program. In developing data flow diagrams in the first step of the design process, it may have already been determined that within a task are several transforms connected by data flows and data stores.

If the decomposition was not taken to this level of detail, a data flow diagram should be drawn for the task. The task is now structured using the Structured Design method. (Depending on the nature of the task, either Transform Centered or Transaction Centered design is used [15].) The structure chart developed for each task identifies the modules in that task and the interfaces between them.

State Dependency in Transaction Processing. A major limitation of Transaction Centered Design is that the action to be taken on the incoming transaction depends only on the input data. In state-dependent real-time systems, the action to be taken depends not only on the incoming data but also on the current state of the system (i.e., on what has happened before).

Yourdon [15] notes that the "difficulty with state-dependent decision procedures is a fundamental defect in the transaction centered structure." The approach proposed by Yourdon consists of distributing transaction processing so that state dependencies are localized. However, in many cases where decision making needs to be centralized, this is not a satisfactory solution.

An alternative is to have one module, a State Transition Manager (STM), maintain both the current state of the system and a state transition table defining all legal and illegal state transitions. A task that needs to process a transaction calls the STM with the desired action as an input parameter.

The STM then checks the state transition tables to determine whether the desired action is legal, given the current state of the system. If the transaction is legal, the STM changes the state of the system, if necessary, and then returns a positive response to the calling task. Otherwise, it returns a negative response. In some designs, it may be necessary for the STM to return a valid action in addition to a positive response (e.g., when the valid action to be taken also depends on the current state of the system).

In DARTS, the STM is designed as a TCM of the IHM type. It maintains a data structure, namely the State Transition Table, which is hidden from the calling tasks. The module also contains the access procedures that check the validity of task requests and perform the state transitions. As with other TCMs, the STM runs in the task that invokes it.

To ensure that state transitions are processed sequentially, they must be mutually exclusive. A good way to ensure mutual exclusion as well as fast state transitions is to increase the priority of the task when the STM is entered and restore the old task priority when the STM is exited.

EXAMPLE OF USING THE DARTS DESIGN METHOD

The DARTS real-time systems design method has been successfully applied to the design of a robot controller system at General Electric's Industrial Electronics De-

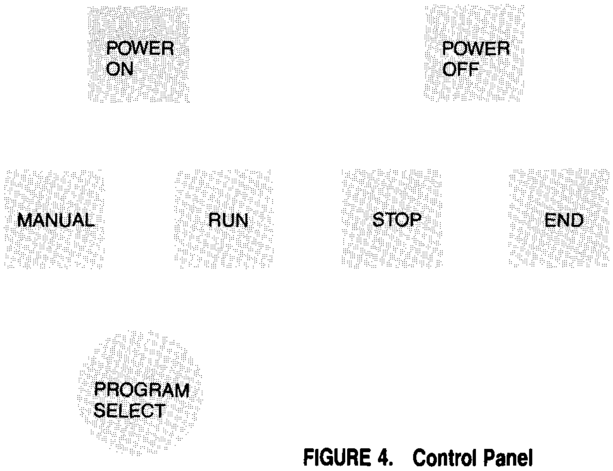


FIGURE 4. Control Panel

velopment Laboratory. The case study presented here consists of a robot controller that controls up to six axes of motion and interacts with digital I/O sensors. Although substantially simplified from the actual design for the purposes of this paper, the case study does serve to illustrate the main concepts of the design method.

Control of axes and I/O is effected by a program initiated from a Control Panel. The Control Panel consists of a number of push buttons and a selector switch for program selection (Figure 4). The state transition diagram for the controller is shown in Figure 5. For reasons of simplicity, error conditions have been ignored.

When the POWER ON button is pressed, the system enters the Powering Up state. On successful completion of the power up sequence, the system enters Manual state. The operator may now select a program using the Program Select rotary switch, which can be set to indicate the desired program number. Pressing RUN initiates execution of the program currently selected, and the system transitions into Running state. Execution of the program may be suspended by pressing STOP, at which time the system enters the Suspended state. The operator may then resume program execution by pressing RUN, returning the system to Running state, or terminate the program by pressing END. Program END having been pressed, the system enters Terminating state; when the program finally terminates execution, the system returns to the Manual state.

Data Flow Analysis

The overall data flow diagram for the robot controller is given in Figure 6. Control panel inputs are read in and validated. Each time a push button is pressed, the input is read and converted to the internal system format by Read Panel Input. Panel inputs are then passed to Validate Panel Input. Since the validity of the inputs depends on the current state of the system, the controller state transition table has to be checked. To keep the example simple, it is assumed that invalid user inputs are ignored.

Valid panel inputs are passed on to Process Panel Input, where they are processed and then passed to the appropriate transform, either Interpret Program Statement or Output Axis Data. In addition, Process Panel Input passes panel outputs (corresponding to control

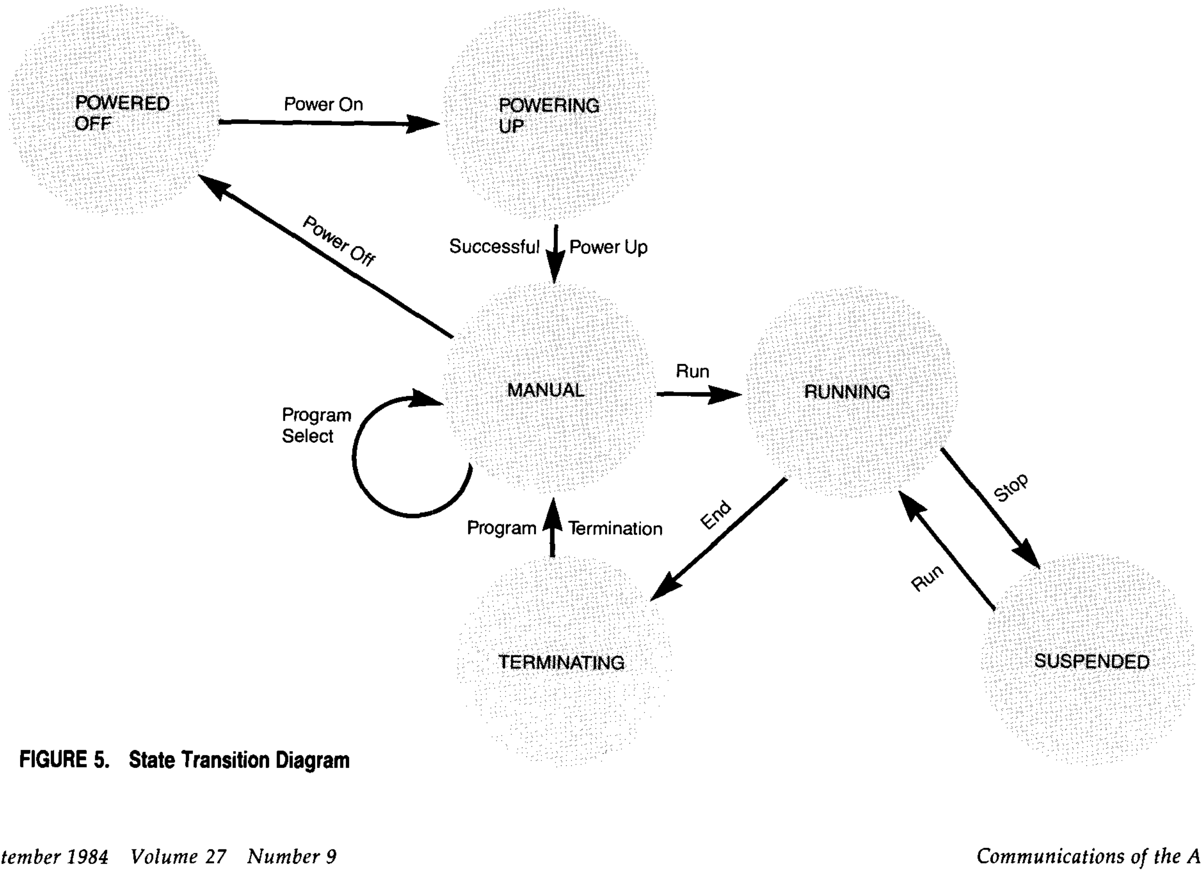


FIGURE 5. State Transition Diagram

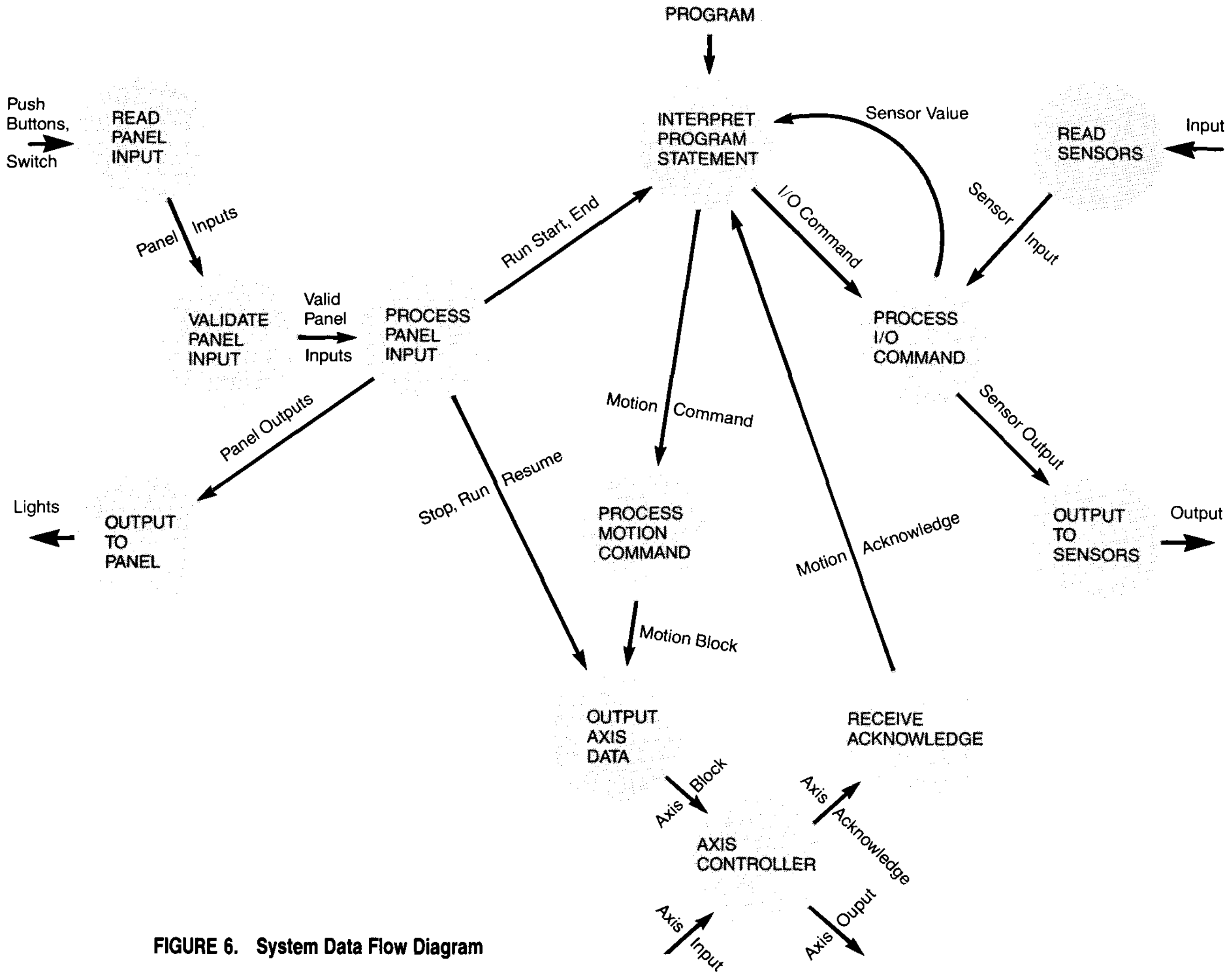


FIGURE 6. System Data Flow Diagram

panel status lights) to Output to Panel.

When the setting on the Program Select switch is changed, the new switch setting is passed to Process Panel Input, which updates the selected program id. When RUN is pressed, Validate Panel Input recognizes, by checking with the State Transition Table, that this is a Run Start. Process Panel Input passes the Run Start request to Interpret Program Statement, which then starts interpreting the program. It executes arithmetic and logical statements directly, but motion and I/O statements require further processing. A motion command is passed to Process Motion Command, which does some mathematical transformations on the data and then passes a motion block to Output Axis Data. Output Axis Data converts the data to the required format for the Axis Controller and passes an axis block to the Axis Controller.

When STOP is pressed, Output Axis Data stops feeding axis blocks to the Axis Controller; when RUN is pressed, it resumes. When the axis motion associated with an axis block has been completed, an Axis Acknowledgment is sent to Receive Acknowledge by the Axis Controller. This acknowledgment is processed and then passed back as a Motion Acknowledgment to Interpret Program Statement.

In the case of a sensory I/O statement, Interpret Program Statement sends an I/O command to Process I/O Command. Process I/O Command receives sensor input data from Read Sensors and passes sensor output data to Output to Sensors.

Structuring the System into Tasks

Having drawn the data flow diagram (Figure 6), we need to consider how the system can be structured into concurrent tasks. Figure 7 shows a box drawn around each transform or group of transforms that logically form a task, whereas Figure 8 shows the system structured into tasks.

As a first task-structuring criterion, typically any function that interacts directly with an I/O device needs to be a separate task since its effective speed is governed by the speed of the device with which it is interacting. Consequently, the Read Panel Input transform needs to be a separate task, the Control Panel Input Handler, since it has to receive inputs from the control panel. Similarly, the Output to Panel transform needs to be a separate task—the Control Panel Output Handler.

The Validate Panel Input transform and the Process Panel Input transform are grouped together into one task, the Control Panel Processor (CPP), in accordance with the temporal cohesion task-structuring criterion. Thus, control panel input is processed immediately after validation.

The transforms Interpret Program Statement, Process Motion Command, and Process I/O Command represent the program Interpreter. As these transforms represent a group of closely related functions, they are grouped together according to the functional cohesion task-structuring criterion. They logically form one task, which may be running concurrently with the CPP.

The Output Axis Data and Receive Acknowledge transforms are grouped together into one task, the Axis Manager, in line with the temporal cohesion task-structuring criterion. Each time Output Axis Data outputs an axis block to the Axis Controller, Receive Acknowledge has to wait for an acknowledgment before Output Axis Data can output the next block. Thus, there is no advantage in having the two transforms execute concurrently. In addition, the speed of these two transforms is dictated by the speed of the axes. Thus, no other transforms can be combined with them into the Axis Manager task.

The Axis Controller is structured as a separate time-

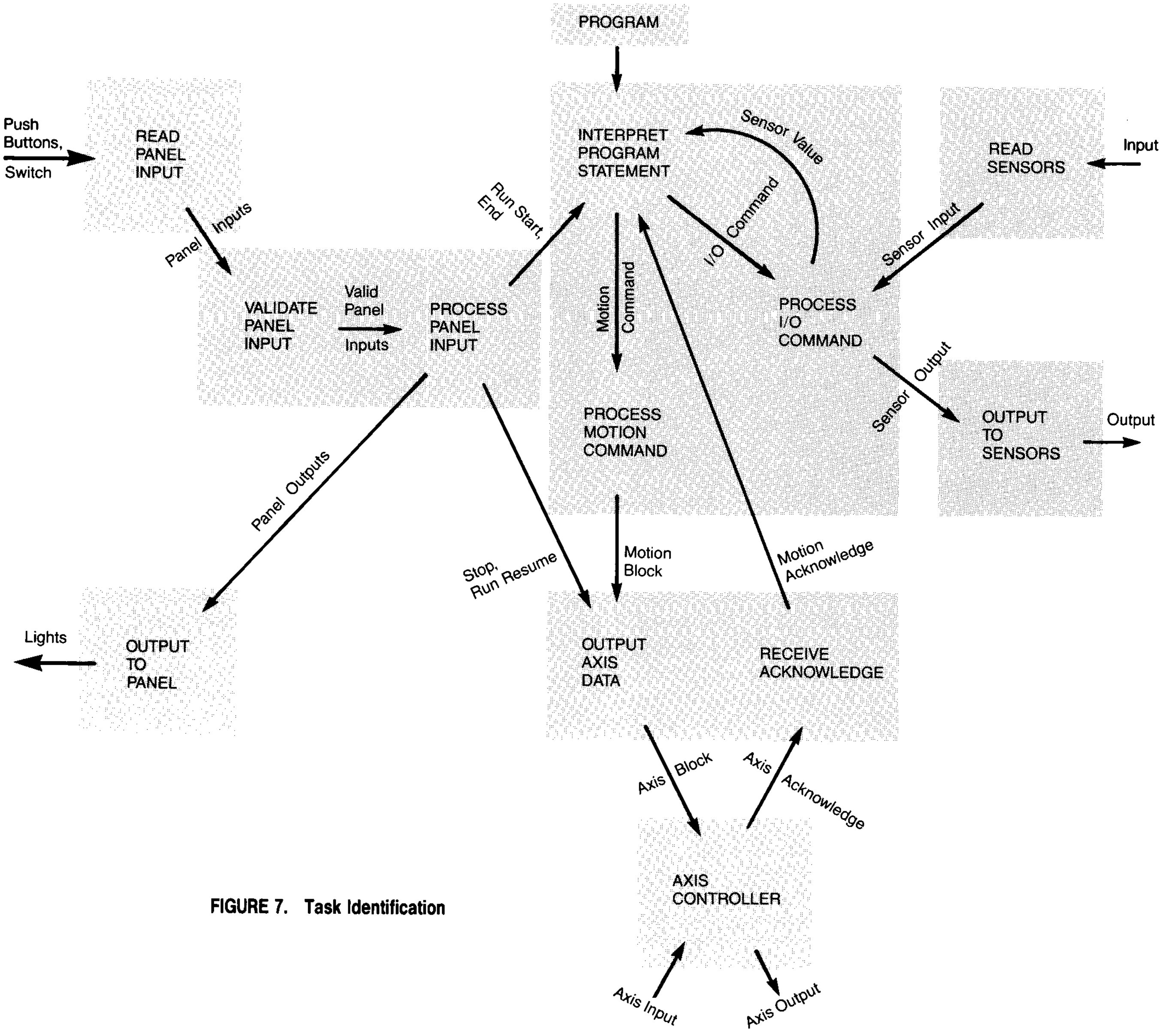


FIGURE 7. Task Identification

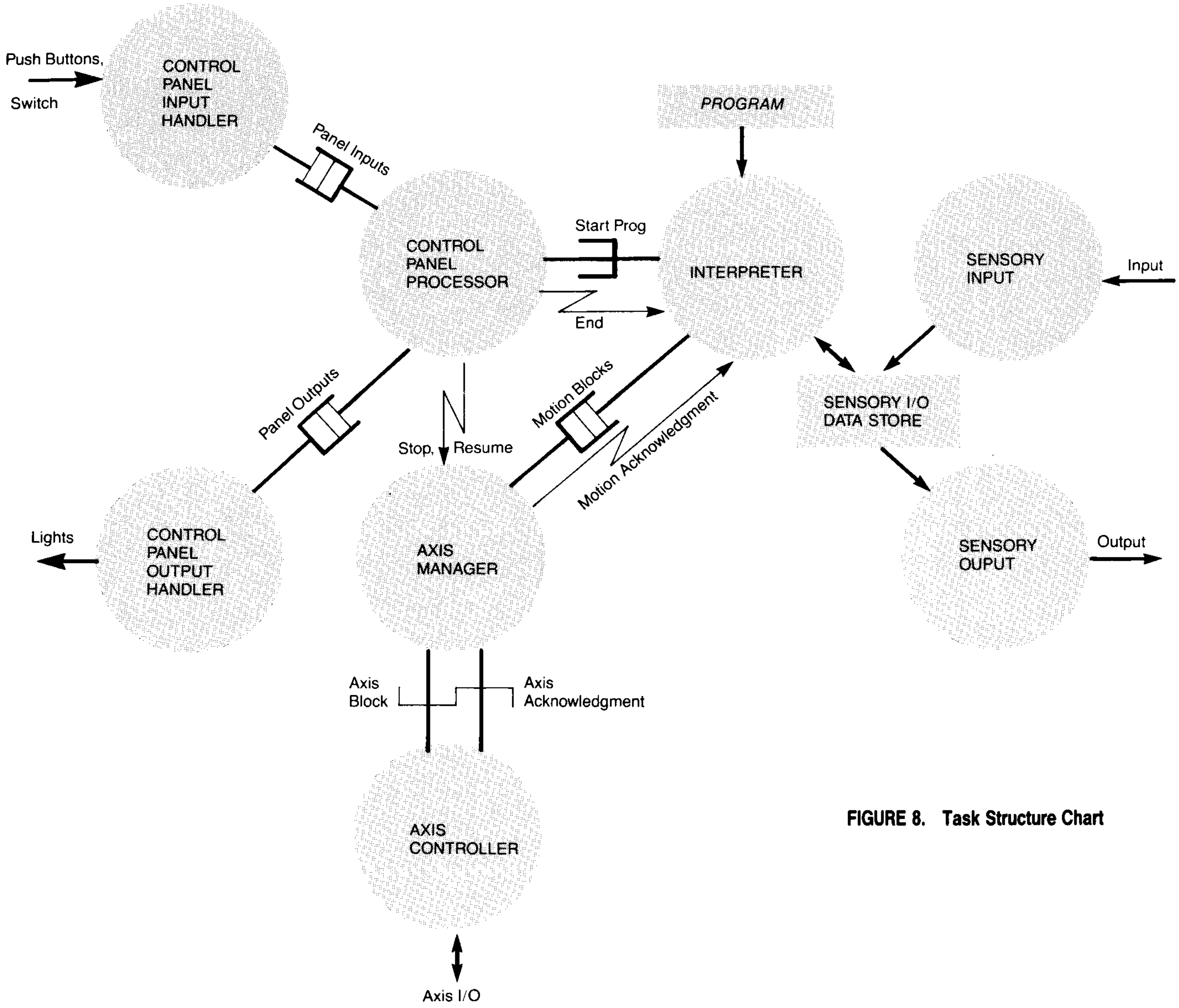


FIGURE 8. Task Structure Chart

critical task. It runs on a separate processor as it interacts closely with the axes.

Sensory I/O requests from the Interpreter are processed by two tasks. The Output to Sensors transform is activated on demand whenever an output is required and so is structured as a separate I/O-dependent task, Sensory Output. The Read Sensors transform periodically scans the input sensors and so is structured as a separate periodic task, Sensory Input.

Defining the Task Interfaces

Once the tasks have been identified (Figure 8), the next step is to define the interfaces between them.

Panel inputs are queued up for the CPP by the Control Panel Input Handler. Thus, the interface between the two tasks consists of a message queue. Similarly, panel outputs are queued for the Control Panel Output Handler by the CPP. An MCM is used for handling message queues.

The CPP sends a Start Program message to the Interpreter identifying the program to be executed. The Interpreter generates motion blocks and places them in the motion block queue. Since some motion blocks imply a long move while others are short, the queue between the Interpreter and the Axis Manager acts as a buffer.

When the Interpreter reads a nonmotion statement (e.g., a sensory I/O command), it needs to wait until axis motion has reached the desired point before executing the statement. The Interpreter waits for a Motion Acknowledge signal from the Axis Manager indicating that all axis blocks have been executed. The Interpreter also waits for an End event signal indicating that the program should be terminated. It is awakened when either of these conditions is set.

The main routine of the Interpreter consists of a TSM in which the Interpreter handles all synchronization conditions. Initially, the Interpreter waits for a Start Program message from the CPP. During program interpreting, it periodically checks to see if an End event has been signaled. When interpreting has been suspended, the Interpreter waits for either an End event or a Motion Acknowledge event.

The Axis Manager receives motion blocks from the Interpreter in its message queue, as well as Stop and Resume event signals from the CPP. The main routine of the Axis Manager is a TSM that handles all synchronization conditions. Every time the Axis Manager waits for a motion block from the Interpreter, it is suspended if one is not available. When the Axis Manager receives the block, it tests to see if a Stop event has been signaled. If so, it waits for a Resume signal. If there is no Stop condition or if Resume was signaled, the Axis Manager sends the axis block to the Axis Controller and waits for an axis acknowledgment of block completion. The communication between the Axis Manager and Axis Controller is an example of closely coupled message communication. An MCM is used to provide the closely coupled communication mechanism.

A sensory I/O data store is used to store the current values of the sensory I/O data. If the Interpreter processes an output command, it updates the sensory I/O data store (SIODS) and signals the Sensory Output task that an output is available. The Sensory Input task periodically scans the input sensors and updates the SIODS when a change takes place. If the Interpreter processes an input command, it reads the SIODS for the current value of the sensor. Since access is made to the SIODS by three tasks, access to the SIODS has to be synchronized by the access routines. Together, the SIODS and the access routines constitute an IHM.

Structuring Tasks into Modules

After the interfaces between tasks have been defined, the next step is to establish the structure of the individual tasks, each of which represents a sequential program. For each task, the data flow diagram is drawn, and from this the structure chart is derived, using the Structured Design method [15].

To illustrate, we will look at one particular task, namely, the CPP. The CPP is an example of Transaction Centered Design supplemented by a STM.

The CPP task shown in Figure 8 was formed by combining the Validate Panel Input and Process Panel Input transforms given in Figure 7. Thus, the data flow diagram for the CPP task (Figure 9) is an expanded form of

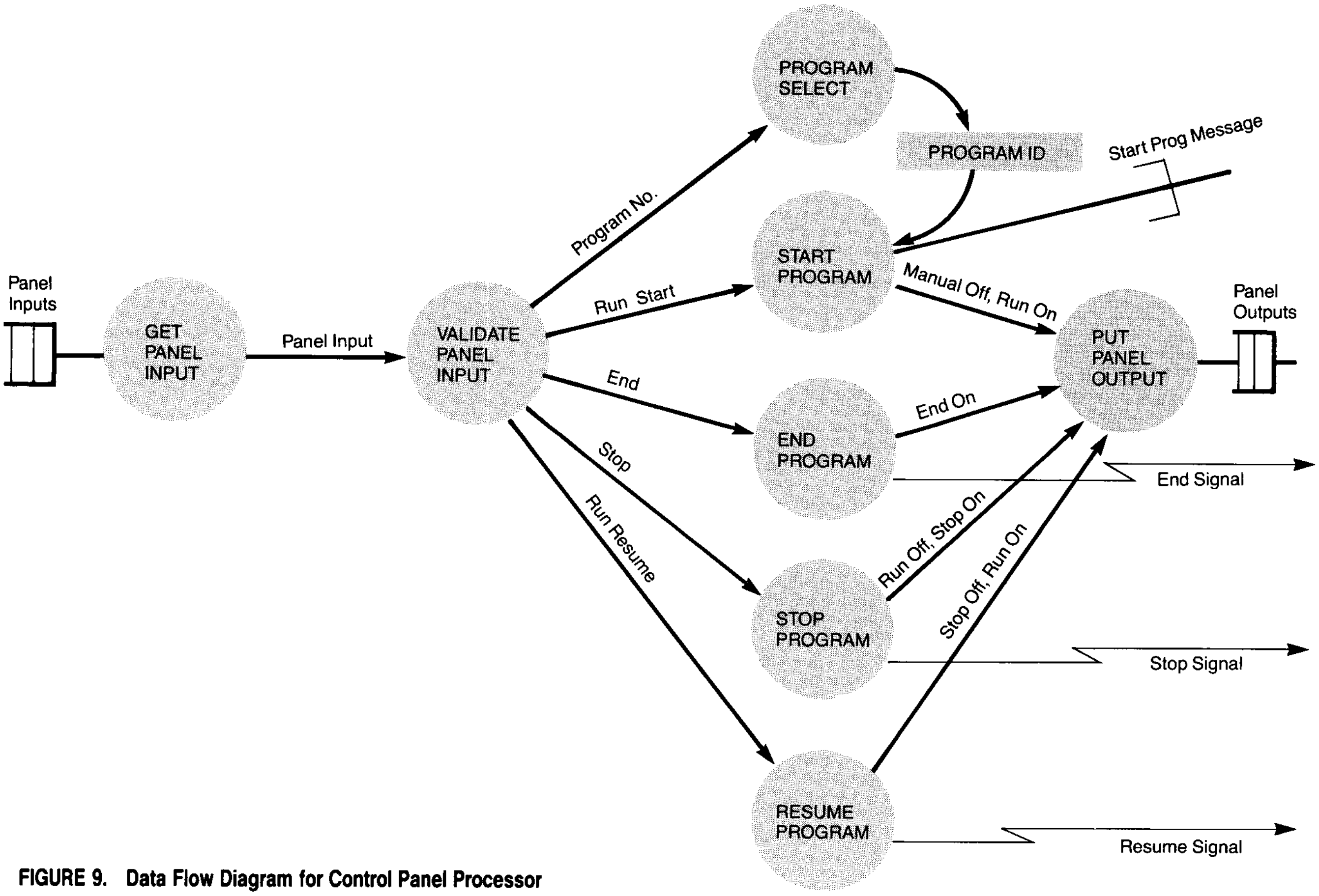


FIGURE 9. Data Flow Diagram for Control Panel Processor

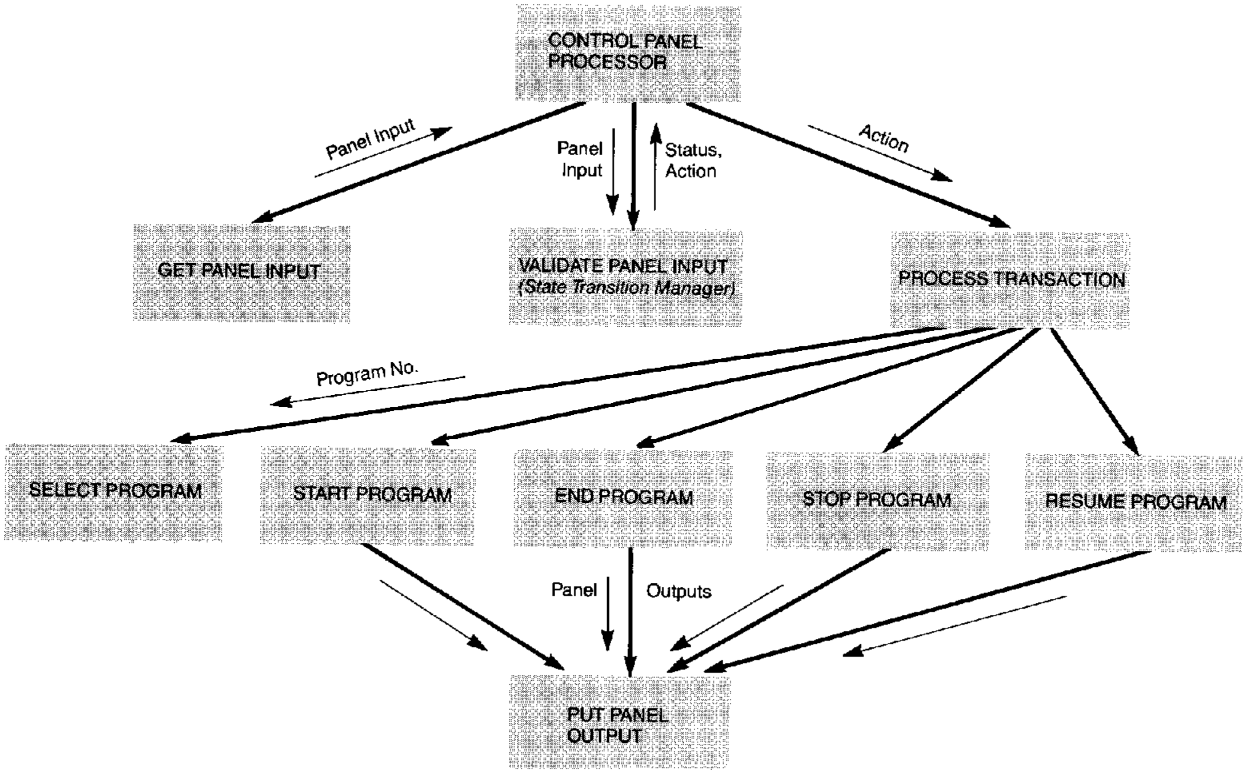


FIGURE 10. Structure Chart for Control Panel Processor

these two transforms.

In Figure 9, Get Panel Input receives input messages from the Panel Inputs message queue. The inputs are passed on to Validate Panel Input, which checks that they are valid for the current state of the system. Assuming the input is valid, the valid input is passed to the appropriate action transform, which performs the action. For example, the Stop Program action transform signals a Stop event, switches off the control panel Run light, and switches on the Stop light. Control panel outputs are passed to the Put Panel Output transform, which queues up panel output messages for the CP Output Handler.

The structure chart for the CPP is given in Figure 10. The main routine, also called the CPP, is a controlling module. It calls Get Panel Input to read a message; if a message is not available, the task will be suspended pending its arrival. When the input message is received, Validate Panel Input is called. The Validate Panel Input module is actually the STM, which is called with the panel input as a parameter. STM returns a Valid/Invalid status.

If the action is not valid for the current state (e.g., STOP is pressed while the system is in Manual state), an invalid status is returned by STM. STM also returns an Action, which is particularly essential where an input may have two or more interpretations depending on the state of the system. Thus, a RUN input identifies a Start Program action if the system is in Manual state and a Resume Program action if the system is in Suspended state.

The control module Process Transaction is now called with the Action as a parameter. Process Transaction calls the appropriate action module to perform the action.

The STM is called by more than one task. Since it is also called by the Interpreter to indicate that program execution has terminated, it is designed as a TCM of type IHM. As with all TCMs, it conceptually runs in the task that invokes it.

EXPERIENCE WITH DARTS

An application of an early version of the design method, before the task structuring concepts had been formalized, is described in [7]. The DARTS design method has so far been used on two projects, a robot controller and a vision system. The robot controller project is now in the system integration phase, whereas the design of the vision system was completed only recently.

There was much discussion initially as to whether DARTS should consider control flow at the start of the design. DARTS, like Structured Analysis, makes no distinction between data and control flow on the data flow diagrams. However, some users of methods like SADT

[13], which do distinguish from the start between data and control flow, often find it difficult to separate data from control early in the design process.

In DARTS, the decision as to which data flows are actually data and which are used for control purposes is postponed to the task structuring stage. That is, control flow is considered at the time that task interfaces are defined and the STM designed. For example, in the case study, the Stop and Resume data flows in Figure 6 become event signals in Figure 8.

In practice, starting the design with data flow diagrams was not found to be a problem, and early concerns about the method were allayed when it was found to work satisfactorily. As in the Structured Analysis method, it was found most useful to postpone considerations of system initialization and error handling to a later stage. In DARTS, this is done at the task structuring stage.

When using DARTS, there is sometimes a temptation to make each transform on the system data flow diagram a task. This usually leads to too many tasks and to unnecessary complexity in dealing with the accompanying communication and synchronization issues. The Task Structuring stage is a crucial phase of DARTS, at which point designers should justify the existence of each task.

While applying DARTS to the robot controller project, a number of significant changes were made to the Requirements Specification after the design had been completed. Although these changes necessitated corresponding changes to the System Design, no major change in the system structure or task interfaces was required. This is attributable to the fact that DARTS leads to a system that is highly modular with reduced coupling between tasks.

At the moment of designing the MCM to handle task communication, a choice existed between using the message communication mechanism provided by the operating system or developing an MCM. The latter approach was chosen for two reasons. First, using an MCM meant that a bound could be set on the size of each message queue in the system. In addition, this made it possible to associate an event condition with each queue. A task could then wait in a TSM for any one of several events to be signaled—a very valuable feature, it turned out, in designing the robot controller.

During the system integration phase, tasks are gradually grouped together and tested. The integration of the robot controller is progressing well, and, up to the time of writing, no major design problems have been encountered. Use of the DARTS design method is considered a significant factor in this smooth integration.

SUMMARY AND CONCLUSIONS

The DARTS design approach described here extends the Structured Analysis/Structured Design method to address the needs of real-time systems by providing an approach for structuring the system into concurrent tasks and for defining the interfaces between tasks. The method leads to a highly structured modular system with well-defined interfaces and reduced coupling between tasks.

Acknowledgments. The author gratefully acknowledges the many stimulating design discussions with J. Lawrence, F. Sherwood, and J. Taylor. The author also wishes to acknowledge the constructive comments made by E. H. Sibley and the referees.

REFERENCES

1. Booch, G. *Software Engineering with Ada.* Benjamin/Cummings, Menlo Park, Calif., 1983.
2. Brinch Hansen, P. Concurrent programming concepts. *Comput. Surv.* 5, 4 (Dec. 1973), 223–245.
3. De Marco, T. *Structured Analysis and System Specification.*Yourdon Press, New York, 1978.
4. Dijkstra, E.W. Co-operating sequential processes. In *Programming Languages*, F. Genuys, Ed. Academic Press, New York, 1968.
5. Gane, C., and Sarson, T. *Structured Systems Analysis: Tools and Techniques.* Prentice-Hall, Englewood Cliffs, N.J., 1979.
6. Gomaa, H. A comparison of software design methods. In *Proceedings of the National Electronics Conference* (Chicago, Ill., Oct.), vol. 33, 1979.
7. Gomaa, H., Lui, J., and Woo, P. The software engineering of a microcomputer application. *Softw. Pract. Exper.* 12, (1982), 309–321.
8. Hamilton, M., and Zeldin, S. Higher order software—A methodology for defining software. *IEEE Trans. Softw. Eng.* (Mar. 1976).
9. Jackson, M.A. *Principles of Program Design.* Academic Press, New York, 1975.
10. Myers, G.J. *Composite/Structured Design.* Van Nostrand Reinhold, 1976.
11. Orr, K.T. *Structured Systems Development.* Yourdon Press, New York, 1977.
12. Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15*, 12 (Dec. 1972), 1053–1058.
13. Ross, D.T., and Schoman, K.E. Structured analysis for requirements definition. *IEEE Trans. Softw. Eng.* (Jan. 1977).
14. Simpson, H.R., and Jackson, K.L. Process synchronization in Mascot. *Comput. J.* 22, 4 (1979).
15. Yourdon, E., and Constantine, L. *Structured Design.* 2nd ed. Yourdon Press, New York, 1978.

CR Categories and Subject Descriptors: C.3 [**Special-purpose and Application-based Systems**]: *microprocessor/microcomputer applications, real-time systems*; D.2.1 [**Software Engineering**]: Requirements/Specifications—*methodologies, tools*; D.2.2 [**Software Engineering**]: Tools and Techniques—*modules and interfaces, top-down programming*; D.4.1 [**Operating Systems**]: Process Management—*concurrency, mutual exclusion, synchronization*; D.4.7 [**Operating Systems**]: Organization and Design—*hierarchical design, real-time systems*; I.2.9 [**Artificial Intelligence**]: Robotics—*manipulators, sensors*; K.6.1 [**Management of Computing and Information Systems**]: Project and People Management—*systems analysis and design*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*software development*

General Terms: Design, Documentation, Management, Standardization

Additional Key Words and Phrases: software engineering, software design, concurrent processes, task communication, task synchronization

Received 12/83; accepted 3/84; revised 5/84

Author's Present Address: H. Gomaa, General Electric, Industrial Electronics Development Laboratory, P.O. Box 8106, Charlottesville, VA 22906.