New S-Band Transmitter Automation Software

W. Stahnke
Radio Frequency and Microwave Subsystem Section

This article describes the status of the 20-kW S-band transmitter automation project. A new software design using a simplified multitasking approach is described that will improve subsystem performance, maintainability and extensibility.

I. Introduction

Previous efforts toward transmitter automation have culminated in the development of a complete, functionally correct program for operating the 100-kW S-band transmitter. This program was developed over an extended period of time as an SRT effort, and it has provided a great deal of insight into the approaches required to automate a transmitter. Structurally, however, the current program could be strengthened. For implementation into the DSN transmitters, it is proposed that a revised program with multitasking be written that should improve subsystem performance, maintainability and extensibility.

II. Rationale for the Design Approach

From the programmer's point of view, the transmitter controller is a general-purpose microcomputer operating a real-time control system. Like many other such systems, it has the following requirements:

(1) There must be a means of responding to events occurring asynchronously in the physical environment.

(2) There must be a means of controlling several devices concurrently.

What sets the transmitter controller apart from some other systems, however, is that it requires very little computational work to be done. Almost all computations fall into the category of integer arithmetic.

In addition, the transmitter control function is simplified by two special features:

(1) Hardware interlocks provide high-speed response to potentially catastrophic events.

(2) Most of the controlled devices are low-speed electromechanical components.

As a result, only moderate response time and moderate control speed are required.

To fulfill the requirements outlined above, a multiprocessing or multitasking approach is preferred. In this case, a single CPU provides ample processing power, so there is little incentive to introduce multiple processors. Because of the abundance of processing power, it is not necessary to make optimal use of the processor itself, so a conventional preemptive, priority-based real-time multitasking executive is not required. Instead, the objectives can be achieved with a simplified executive that distributes processor time uniformly among the various tasks. This approach brings the advantages of multitasking to the program without the usual complexities of a full multitasking executive.

This simplified executive function can be depicted by the top-level flow chart shown in Fig. 1. After power-on initialization, the first task is executed for a given period of time, typically one millisecond. After that time has expired, execution of the first task is interrupted by a hardware timer, and the machine state is stored in a corner of memory reserved for that purpose. The second task is then executed for an identical period, until it is interrupted. After the last task is interrupted, the machine state is restored in preparation for resuming the first task. This cycle is repeated indefinitely.

It is important to note that in the strictest sense Fig. 1 is not a flow chart at all. Figure 1 implies that (for example) task 2 is initiated when task 1 is finished and relinquishes control of the processor. In fact, task 1 is interrupted by an event external to it, and later resumes execution at the same point at which execution was previously interrupted. The code for task 1 (and, in fact, the code for all of the tasks) is written as an endless loop, so that it is never done.

III. Advantages of Simplified Multitasking

The simplified multitasking approach outlined above has several advantages over the polling-loop approach to control systems, including:

(1) Concurrent execution of several tasks.

(2) Faster response to external events.

(3) Programmatically invisible processor allocation.

The last point is so important that it deserves a detailed discussion, because it is the principal reason for using multitasking.

In control systems, there are often some functions to be performed that require very little processor time, but require a great deal of elapsed time to execute. One example is filling a tank. The processor must open a valve and wait until the tank is full before closing it. The elapsed time might be several hours.

If one were to write a conventional sequential program to perform this function, the processor would open the valve and then enter a short testing loop to determine if the valve should be closed (see Fig. 2). This simple program might occupy the processor full time for hours, during which time no other functions could be performed (including operator intervention to abort the filling). This is clearly unacceptable.

The only alteration that can provide better performance using a polling-loop approach is to rewrite the program in such a way that it can be called repetitively by the (higher-level) polling loop. This usually necessitates the use of state flags to allow the processor to determine what it was doing on the most recent pass. For our simple example, the program might be rewritten as shown in Fig. 3. Here only one state flag is required.

The modified routine provides improved performance at the cost of simplicity and clarity. It is now the programmer's responsibility to break the task into subtasks, each of which must execute quickly enough to prevent undue delay to the polling loop. The program and its flowchart have become more difficult to read.

At the most basic level, the difference between the sequential code and modified sequential code is quite simple: in the former, the state of the system is implied by position in the code; in the latter, it is explicitly given by state flags, which must be defined and manipulated by the programmer. The programmer must also be aware of the timing constraints that each routine imposes on the polling loop.

A completely different solution is possible by introducing multitasking. The program can now be written using purely sequential coding techniques, and the multitasking executive allocates processor time to each task automatically. Each task is written as a free-standing sequential program, without regard to polling time. A task is concerned only with itself; the other tasks are invisible to it. This greatly simplifies program design, coding and debugging.

It should be clear at this point that introducing multitasking does not increase the complexity of a real-time program; on the contrary, it enables the programmer for the first time to write simple, purely sequential routines for each application task.

The simplified multitasking approach presented above also has some advantages over conventional preemptive priority-based multitasking. Among these are:

(1) The executive is simplified.

(2) Fewer interrupts are required.

IV. Functions of the Executive

The executive performs two separate functions: initialization and multitasking. It also provides a global real-time clock for use by all of the tasks. These functions are described in greater detail in the following paragraphs.

Initialization by the executive encompasses both hardware and software initialization. Hardware initialization is necessary because on power-up the previous state of the hardware is lost, and the hardware must be set to a known state before multitasking can occur. Software initialization includes setting the initialization semaphore and initializing the storage area for each task.

The initialization semaphore is a single byte whose value is set after power-up to the number of tasks. Each task has associated with it an initialization routine of its own that may require an indefinite amount of time to execute. After finishing its initialization, each task decrements the initialization semaphore and then enters a testing loop that waits until the value of the semaphore is zero. In this way, all of the tasks complete their initialization before any task begins to execute working code. This is shown in the flowchart of Fig. 4 for a single task.

The storage area associated with each task (Fig. 5) must be partially initialized before task rotation can occur. A minimum of two 16-bit register images must be filled: the program counter image and the stack pointer image. Each program counter image must be set to the address of the first instruction of its corresponding task initialization routine. Each stack pointer image must be set to the ceiling value of the stack area reserved for the executive. This is necessary because task rotation requires the existence of a user stack.

The global real-time clock is maintained in memory, and must be reset to zero after power-up. Subsequently, its value is available to all of the tasks for timing physical events. The clock is incremented every time control returns to the executive from a task. The initialization sequence is shown in Fig. 6.

Task rotation is initiated by a hardware timer interrupt. There is a separate task rotating routine for each task. Task rotation is straightforward. The register contents are stored in the register images of the previous task storage area. After updating the interrupt dispatch and global real-time clock, the registers are restored from the next task storage area. Control is then passed to the next task. This sequence is shown in Fig. 7.

V. Present Status

Design and coding of the 20-kW S-band transmitter controller program is proceeding using the multitasking approach outlined above. The executive has been completed, and the hardware drivers are currently being written. Later reports will cover testing of the controller with the simulator for evaluation and support of unattended operation demonstration.

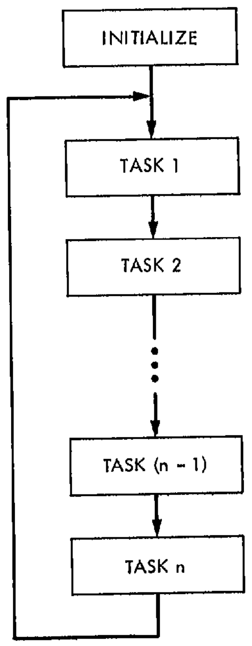

Fig. 1. Executive top-level flowchart

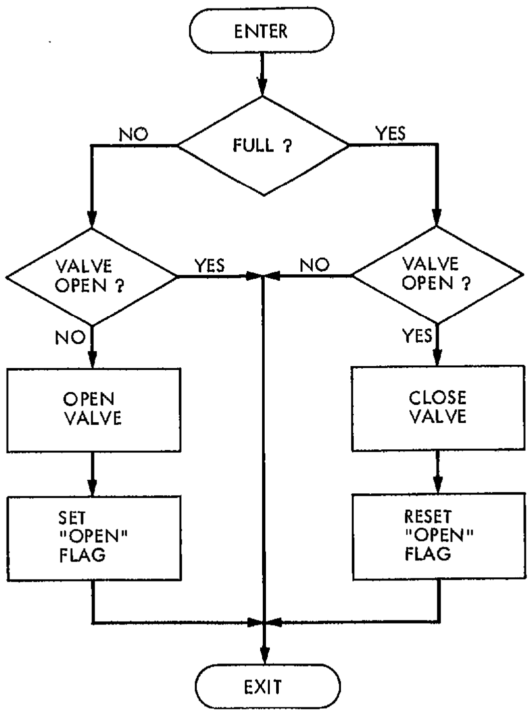

Fig. 3. Valve control flowchart (modified sequential code)

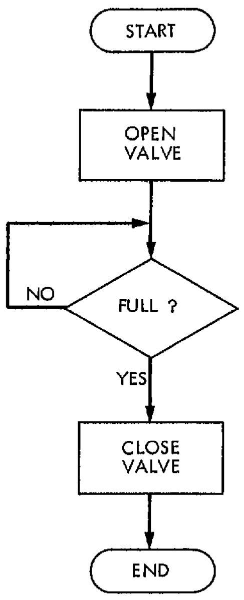

Fig. 2. Valve control flowchart (sequential code)

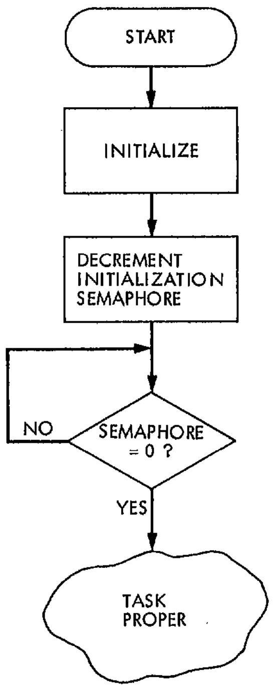

Fig. 4. Task initialization flowchart

H REGISTER	L REGISTER
D REGISTER	E REGISTER
B REGISTER	C REGISTER
ACCUMULATOR	FLAGS
STACK POINTER	
PROGRAM COUNTER	

Fig. 5. Register image storage area for a single task

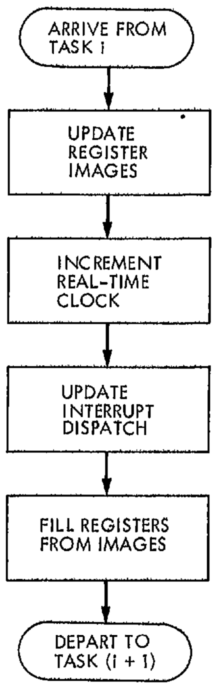
ARRIVE FROM TASK i
UPDATE REGISTER IMAGES
INCREMENT REAL-TIME CLOCK
UPDATE INTERRUPT DISPATCH
FILL REGISTERS FROM IMAGES
DEPART TO TASK (i + 1)

Fig. 7. Task rotation sequence (typical)

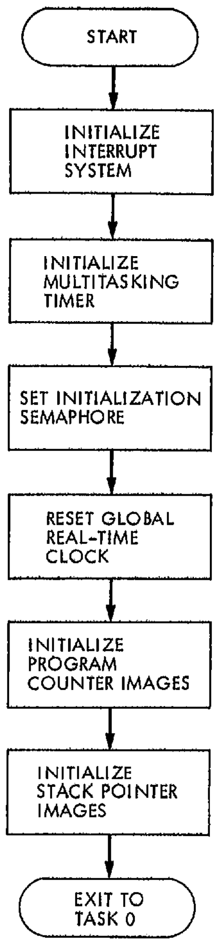
START
INITIALIZE INTERRUPT SYSTEM
INITIALIZE MULTITASKING TIMER
SET INITIALIZATION SEMAPHORE
RESET GLOBAL REAL-TIME CLOCK
INITIALIZE PROGRAM COUNTER IMAGES
INITIALIZE STACK POINTER IMAGES
EXIT TO TASK 0

Fig. 6. Executive initialization sequence