Management of periodic operations in a real-time computation system

by HENRY WYLE and GERALD J. BURNETT
Autonetics
Anaheim, California

INTRODUCTION

I. Presentation of problems

To understand the problems of real-time control systems it is beneficial to review briefly some of the characteristics of command and control or commercial multiprogrammed systems. Typically, these systems are not time critical and as a result can schedule programs on a queue basis. The System Executive's master scheduling program keeps track of operational programs ready to run (and also keeps track of free processors if the system is a multiprocessor) and then simply assigns programs running time on a priority basis. If the processor (or processors) happens to be busy at a particular moment, the ready-to-run program list becomes a queue. Thus the timing relationships among operational programs of lesser priority are somewhat random, and there is a somewhat unpredictable wait for a given program. In many applications this random wait is acceptable; however, in many real time control systems, in particular real time avionics and space systems, this randomness is not acceptable. For example, if it is time to execute a program with a precise periodicity requirement there must be a guarantee that the processor (or a processor) is available or has lower-priority, interruptable programs in execution. If all the processors happen to be engaged in executing other programs with precise periodicity requirements at this time, a system bottleneck would exist. This bottleneck would introduce errors in the accuracies of the computations. For example, in an avionics system, if the periodicity is not precisely held, a weapon delivery program could easily cause the weapons to miss the target, or an automatic terrain-following program could cause a plane to crash or be forced to pull up to higher altitudes.

In the past, periodic programs were generally scheduled by using a real-time clock interrupt to call the highest rate periodic program in a system. The succeeding periodic programs were then brought into execution by linking them to other periodic programs with control words. This approach typically required that all the programs take a fixed amount of execution time regardless of internal branches and that almost all but the highest rate periodic program be run at rates that are higher than necessary. This inefficient use of the processor (running programs faster than necessary) has been lessened in more recent systems by varying the linkages depending on the real-time clock interrupts (sequencing a number of different programs in the time periods between interrupts). The number of interrupts can be counted by an executive program and groups of internally linked programs executed periodically every n^{th} interrupt. However, time is still wasted monitoring the clock and waiting for a program's execution time to occur. A more important inadequacy of these latter program schedulers is that they are inflexible to program changes. In particular, if a number of program changes occur, the programmer must rework many of the linkages, initializations, and time constraints by hand.

A scheduler was developed to guarantee the availability of a processor (the processor) whenever a periodic program is to be executed, to save the processor time that was wasted by the linkage scheduling method, and to provide ease of automatically handling program changes. This scheduler, described in Section II, uses a real-time clock, an ordered table of the periodic programs, and a subroutine to execute the periodic programs at their precise rates.

The subroutine (or the scheduling algorithm) is called the Local Scheduler because, in a multiprocessor* system, processor-memory combinations

*This statement and those which follow apply to a multiple-computer system as well as to a multimodule multiprocessor as in Figure 1.

are set up so that each uses a Local Scheduler and a local grouping of periodic programs. A Master Scheduler also exists in this type of system (in a single-processor system the Master Scheduler is only part of the Local Scheduler) to schedule nonperiodic programs, but it would be less efficient than the Local Schedulers at carrying out the scheduling of a large number of periodic programs. A discussion of this lack of scheduling efficiency for a Master Scheduler is presented at the end of Section II.

The periodic programs mentioned have a number of real-time sensors associated with them; as a result, because of data collection and communication delays, it is generally a difficult problem to have fresh I/O data available to a periodic program as soon as it goes into execution. Typically, a decision must be made between: (1) managing these sensors in an asychronous manner by sampling and placing them in memory at some multiple of their normal rate, or (2) managing these sensors in a synchronous manner by sampling them at the proper rate by program-controlled requests. However, either of these methods present some problems. Managing the sensors in an asynchronous manner avoids letting the information become too old between processor accesses, but it means that it is necessary to handle moderate I/O rates as if they were actually high I/O rates. As a result, extra buffering hardware and timing circuitry must be added to the computation system. If the sensors are managed in a synchronous manner, a processor might spend a considerable amount of time waiting for I/O variables that are needed very early in a program (the program is loaded and then cannot begin until it receives a number of variables from sensors). This wait could easily range to many milliseconds; as a result, the wait for I/O variables could easily waste a portion of a processor's time. From the foregoing, we can see that one way to manage the I/O traffic at a moderate rate and yet waste the minimum amount of processor time waiting for the variables is to give the I/O system the facility to call I/O variables just before the associated program goes into execution. This look-ahead feature was implemented in the Local Scheduler, and is described in Section III of this paper.

Another problem, changing the periodic program sequences, exists in managing periodic programs in a multiprocessor system that must be reconfigured after module failures. (Again, a good example is a central computation system in an avionics or space system.) Such a multiprocessor system is shown in Figure 1. All modules of the same type are exactly alike so that the highest priority computations can be continued after a number of failures as long as there is one of

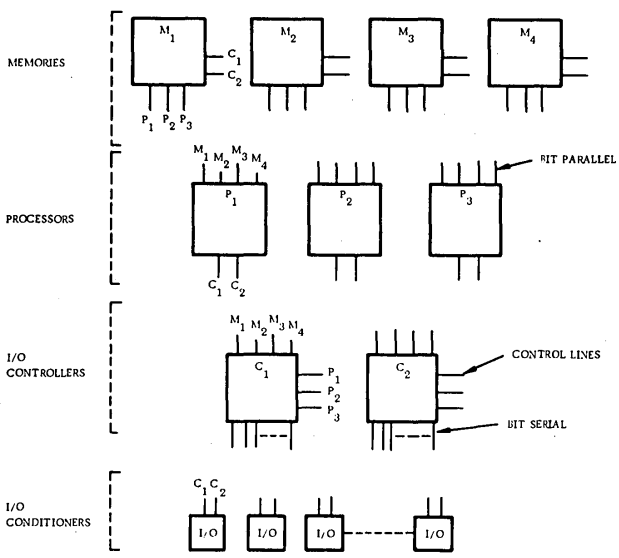

Figure 1 – Multiprocessor

each type of module present. After a failure (or failures) there is less computation power available; consequently some of the programs will be eliminated. This means that the periodic programs in a reconfigurable multiprocessor must be managed so that the setting of new program sequences is a simple task. Because the Local Scheduler was initially developed for a reconfigurable multiprocessor (shown in Figure 1), it can easily adapt to program changes. This is explained in Section II.

In summary, the Local Scheduler presented in Section II provides for guaranteed scheduling of periodic programs at their precise rates, relieves the programmer of the necessity to link programs to each other, provides for early call of I/O variables if necessary, and enables easy adapting to program sequence changes. As will be shown, this is all done with only a small overhead cost for execution of the Local Scheduler subroutine.

II. Local scheduler

As discussed in Section I, the Local Scheduler is an algorithm that uses a subroutine and a real-time clock to schedule periodic programs on a processor-memory combination. (This refers to one of the processors and memories in a multiprocessor system or to the only processor and memory in a single-processor system.) The Scheduler operates so that when the periodic programs have been completed the Executive (Central Scheduler) is notified that the processor and memory are free to do background programs.

The Local Scheduler will then interrupt the background programs and also notify the Executive when it is again time to carry out its periodic programs. The Scheduler also relieves the programmer of the task of specifying execution sequences or program linkages within the operational programs themselves. This job is not only time-consuming but also inefficient if not impossible when a large number of programs of varying periodicity must be interleaved on one processor. As mentioned earlier, a third useful property of the Local Scheduler is the ease of executing operation mode changes and of adapting to program changes either because of reconfiguration after a malfunction or because of a future system change. These points will be clarified by the following explanation of the Scheduler operation. The discussion of the early call of I/O variables is presented in Section III.

To best explain the Scheduler's operation, we will present an example. Let us assume a group of programs with periodicity requirements, as in Figure 2. Assume that the first three programs, A, B, and C,

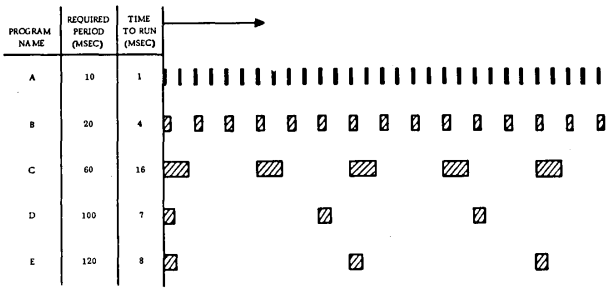

Figure 2 — A group of programs with periodicity requirements

have precise periodicity requirements and the last two programs, D and E, have only approximate periodicity requirements. The total running time of the whole program group is about 700 msec per second, so that it can theoretically be accommodated on one processor with 30 percent execution time to spare (this time can be used to work on a queue of background programs from the Master Scheduler). One possible interleaving of these programs for a single processor is shown in Figure 3. The rule used to interleave the programs is: Start out doing the most frequent program, A. Upon completion, do B, then C, D, etc. When it is time to do A again, interrupt what is being done and do A; then see if it is time to do B again, if so, do B. Keep going down the line until the interrupted program is reached and then resume it. In trying to get back to the interrupted program, several of the programs higher on the list may have to be completed. Thus, several nestings will sometimes occur. One property of the algorithm may be seen from Figure 3. Programs successively lower in the list are guaranteed precise periodic execution as long as their period is an integer multiple of the period of the program immediately above them. Whenever this rule is broken, the program breaking the rule and all programs below it can expect their execution period to be perturbed randomly, although their average execution period will remain the desired one. Thus, the chart in Figure 2 would tell us that Programs A, B, and C will have precise periodicity, and Programs D and E will be perturbed. The timing chart in Figure 3 confirms this. This property of the algorithm places no unreasonable constraint on the system. It simply says that if precise periodicity is desired the programs should be adjusted to meet the integer multiple rule.

The local scheduling algorithm is simple to implement with a real-time clock in the processor and with a subroutine. The real-time clock is set to the period of the fastest program (A). It interrupts, saves the program status, gets reinitialized, and always transfers program control to A. When A or any of the other programs is finished, the finishing program always transfers to the subroutine (the Local Scheduler). The principal function of the Scheduler is to search through a table that defines the Program Group (see Figure 4) and determine which program is next. The state of the table is shown at time $t_a = 271$ msec in Figure 4. "A" has just been completed, and the processor's real-time counter says 271; therefore, Program B will be run next. The explicit operations performed by the Scheduler Subroutine are shown in the flow chart of Figure 5.

To determine the speed of the Local Scheduler, the Scheduler was programmed using a simple instruction repertoire. Assuming a memory cycle time of 2 microseconds, the Local Scheduler consumes approximately 0.6 percent of processor time per 100 programs per second executed by a given processor. The other portion of the Local Scheduler is the real-time clock interrupt routine. It is executed at the rate of the highest rate periodic program (100/second in the case of Figure 2) and consumes about 80 usec per execution or 0.8 percent of processor time per 100 executions per second of the highest rate program. This means that the total Scheduler overhead for the example of Figure 2 is only 2 percent.

We can now specify the means whereby the scheduling operation is updated after a system reconfiguration or a change in the computation load. A system reconfiguration, discussed in Section I, is necessary in some advanced multimodule systems (see Figure 1) where it is generally desired to maintain a high relia-

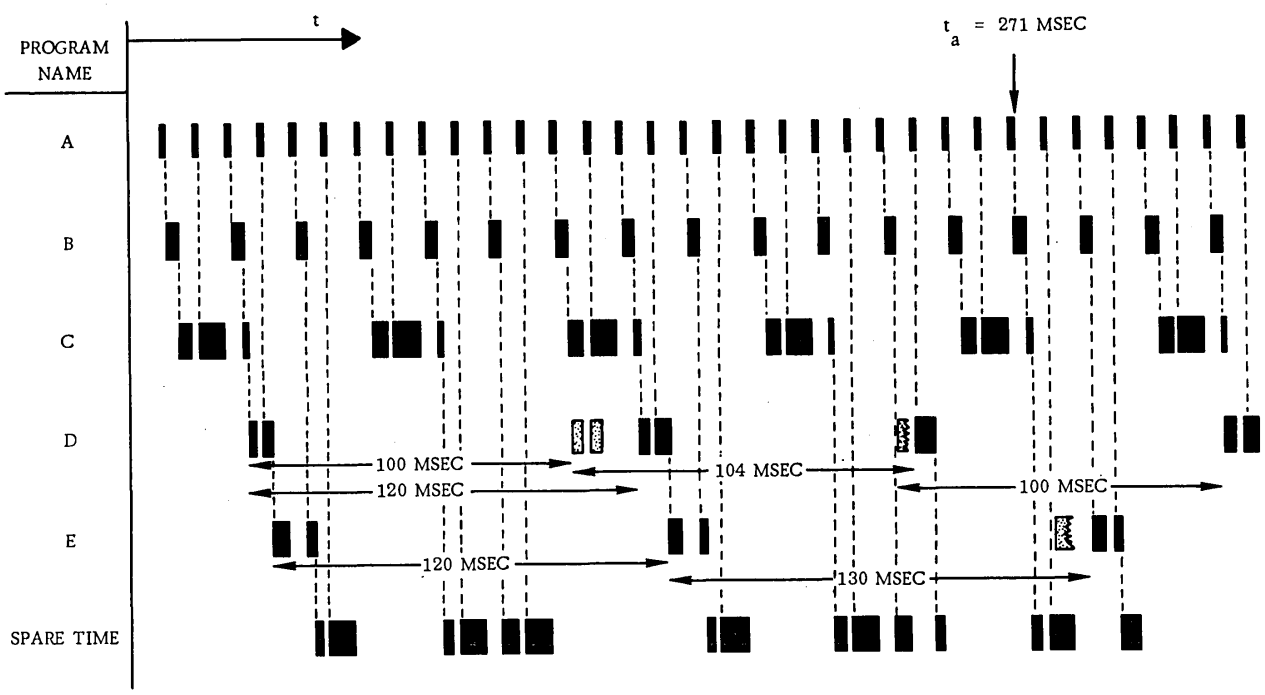

Figure 3 – A practical side for a periodic program group

REAL-TIME COUNTER: 1.00....00271

"NEXT" (NEXT TIME EXECUTION OF THIS PROGRAM IS TO BE BEGUN. SIGN BIT IS ALSO USED AS NONCOMPLETION FLAG.)	"PERIOD" (PERIOD AT WHICH PROGRAM IS TO BE EXECUTED) (MSEC)	"ENTRY" (PROGRAM STARTING LOCATION)	"RESUME" (PROGRAM RESUMPTION LOCATION)
---	10	(A) --	
1.00....00271	20	(B) 4110	--
0.00....00255	60	(C) 3170	3415
1.00....00347	100	(D) 2663	--
1.00....00285	120	(E) 5111	--

Figure 4 – Program group table of the local scheduler

bility for a certain set of computations. For example, it would be desirable to continue the navigation calculations in an avionics or space mission after a number of failures to be able to return home safely. In this case the Master Scheduler would have a table with the necessary navigation programs marked. After any failures it would be necessary to change the program sequences so that at least the navigation programs are being carried out by an active Local Scheduler.* Changes in the computation load can occur at any time in multiprocessor or single-processor systems because of new or unforeseen needs. These changes would also require changes or additions to the existing Local Scheduler program sequences. It should now be noted that both of the aforementioned operations affect the scheduling of periodic programs in the same manner, because in both cases it is necessary to set up a new program mix on the processor memory pair. The Executive Program could simply go into the Program Group Table and remove the rows devoted to the terminated programs and close up all the rows. A comparison is then made between the period of the program (or programs) to be brought in and those on the table. The new program is then placed between the two rows that bracket its period. (Alternately, bypass bits on particular table rows could be turned on and off.) The Local Scheduler is then restarted with the highest rate program executed first. It should be noted that this operation is very simple and can be carried out automatically by the

*The reader may ask why a change in program sequence would be necessary because this group of programs could have been set up together initially; however, failures early in a mission may require some portion of the navigation programs and the automatic terrain-following program to be continued in the remaining Local Scheduler. As a result, these programs may be set up together initially.

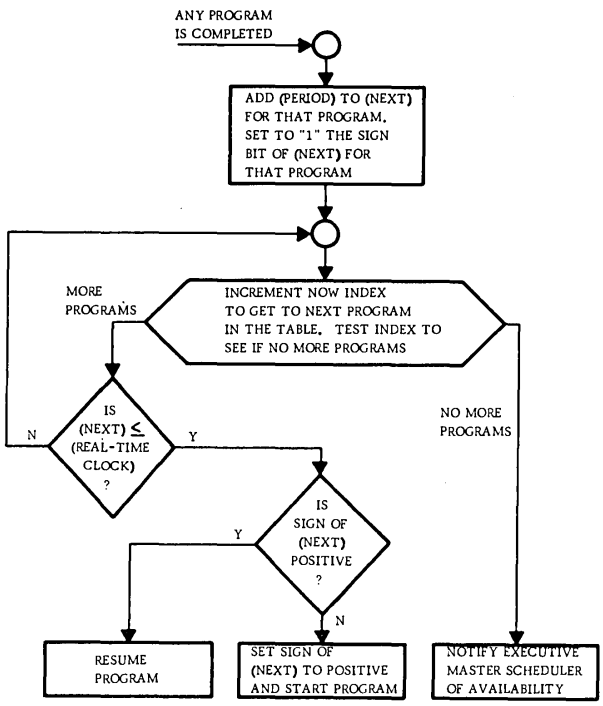

Figure 5 – Flow chart of the local scheduler

System Executive. This should be contrasted with the problems of changing linkages in a scheduling system that uses control words to link periodic programs to each other.

Having presented the Local Scheduler, it is worthwhile to briefly compare it to a Master Scheduler. Of course, a Master Scheduler is only distinct from the Local Scheduler in a multiprocessor system. In this case this large scheduler would be executed by one processor and would schedule the periodic programs for all the processors. To avoid making the system have single special modules (a single failure point), each processor module would have a real-time clock and be capable of executing the Master Scheduler; as a result the Master Scheduler saves no hardware. The Local Scheduler, as pointed out, is executed whenever a real-time clock interrupt occurs or when ever a program is completed. Therefore, the Master Scheduler processor would save the other processors the time for each to execute the real-time clock routine, but each processor would sit idle while the Master Scheduler performs its program completion and program initialization functions. Because these latter functions take the majority of the time, the Master Scheduler would increase the scheduling overhead. An even worse problem could easily arise with the Master Scheduler if two programs were to complete at once. In this case one of the two following programs would be started late. The scheduler would need to become very involved to avoid the latter problem. The need to have the Master Scheduler interruptable at all times would also inhibit the scheduling processor from carrying out precisely periodic programs. Thus the Local Scheduler is not only a very efficient scheduler for single-processor systems, but also for multiprocessor systems.

III. Early call of I/O variables

As discussed in Section I, the Local Scheduler can be used to manage I/O traffic for the periodic programs efficiently. It carries out this function by calling I/O variables for a periodic program just before it is to go into execution. In this way three important goals are achieved:

1. Separation of Computation from Input/Output
   To preserve the integrity of multiprecision numbers and of parameter sets that are internally related, it is essential to avoid computation on data in memory during the time interval in which that data is input or output.
2. Control of Transport Lags Through the Computer System
   Both uncertainties as to transport lags (minimizable by separating computations from input/output) and excessive transport lags (minimizable by establishing the proper time sequence between input, computation, and output) must be controlled.
3. Placing Input/Output Delays Off-Line from Computation
   Where input/output must take place in connection with format conversion and/or data communication, it is frequently prohibitively wasteful of computer time for the computer program to request input parameters and then halt until they arrive. Thus some sort of off-line procedure resulting in the data being available to the program when the program needs it must be used.

One possible implementation of this task is described hereinafter. This implementation may be referred to as "NESTED."

To have the Local Scheduler also carry out the early call of I/O variables, two special input timing registers, the High Rate Register and Low Rate Register, can be added to the processor. The system then operates in the same manner except for the inclusion of one new task for the Executive and another for the Scheduler.

The task for the Executive involves initially setting each processor's High Rate Register to a time delay appropriate to the sensors associated with the

highest execution-rate program of a particular program group, e.g., 500 usec. Alternatively, this register could be eliminated by the choice of an appropriate, fixed time delay. When the real-time counter in a processor has decremented to the value contained in its High Rate Register, a high rate request is automatically sent to the proper controller. The controller accepts the request (as soon as possible), goes to a preset memory position, and picks up and executes the high rate I/O program. In this way, the processor is able to immediately begin work on the high rate program when the system is interrupted for this purpose.

The new task for the Scheduler, to enable it to help call I/O variables, involves updating the Low Rate Register whenever a program is completed. The Scheduler still does its normal job of choosing the next program, but it is now also required to compare the next execution time of the program following the next program to be run and the second from top program in the Scheduler table (Program B in Figure 4). Whichever of these two programs is to be executed first has its next execution time, less some delay (say 500 usec), placed in the Low Rate Register. The Scheduler then sets up the proper memory word with the location of the I/O program for the chosen program's sensors. This is accomplished by loading an established memory position with the program starting location from the Entry column of the Local Scheduler Table, Figure 4. The controller, when requested by the low rate interrupt, goes to the established memory position and obtains and executes the I/O program. After the aforementioned extra task, the Local Scheduler as before sends the processor to the next program to be run. The explicit operation of this portion of the Scheduler is shown in the flow chart of Figure 6. This figure is a continuation of Figure 5.

The preceding I/O call routine was also programmed to determine its effect on the execution time of the Scheduler. Again, assuming a 2-microsecond memory cycle, the Local Scheduler was increased, from approximately 0.6 percent of processor time per 100 programs per second to approximately 1 percent of processor time. With the foregoing overhead, plus the 0.8 percent per 100 executions per second for the real-time clock routine, the total scheduler and I/O variable call overhead can be calculated for any system. For example, in an Avionics system under study, one processor carries out the majority of the periodic programs. This amounted to about 700 programs executions per second of which the highest rate program accounted for 200 executions per second. Therefore, this heavily loaded processor would only spend less than 9 percent of the time in overhead.

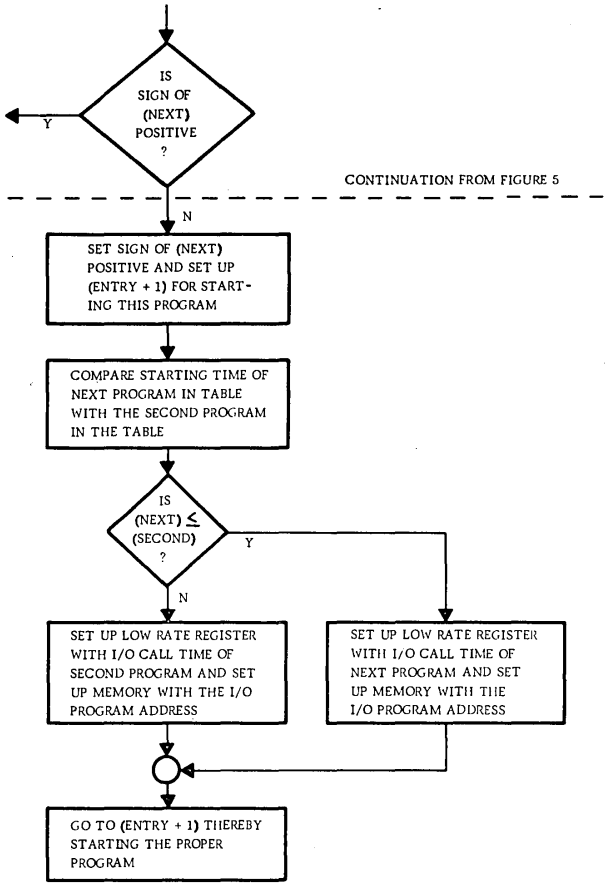

Figure 6 – Flow chart for the I/O call routine

An alternate and somewhat simpler I/O traffic management scheme can be used. This scheme, which we shall refer to as "QUANTIZED," does not require the High Rate and Low Rate Registers of the NESTED scheme, and it leads to a simpler Local Scheduler.

One difference between QUANTIZED and NESTED is that iteration rates of computations in QUANTIZED must be submultiples of the next-higher iteration rates. Thus, in Figure 2, Program D would have to be scheduled with a period of either 60 or 120 milliseconds, rather than 100 milliseconds.

A second difference between the two schemes relates to the granularity of starting times for I/O actions. In NESTED, computations can start at irregular times. The sensor inputs to these computations can be initiated (by the High and Low Rate Registers) at more or less arbitrary times prior to the scheduled computation starting times. Because the computation starting times need not be correlated with each other, neither will be the various I/O starting times. This can result in a controller executing I/O sequences nested within other I/O sequences, and can also result

in uncertainty as to the starting time of some I/O sequences. Such uncertainties of I/O execution diminish the efficiency and speed of I/O execution.

In QUANTIZED, I/O sequences can be started only upon a processor real-time counter interrupt. Furthermore, because the iteration rate submultiple method for computations leads to an easily predictable computation sequence (especially if binary submultiples are used), a corresponding predictability is associated with I/O sequence execution times. This is related to the elimination of nesting of I/O sequences in other sequences. The most practical technique for QUANTIZED has been found to be the following:

1. Group together I/O actions for computations of the same iteration rate into one long I/O sequence, or into one long Input sequence and another long Output sequence.
2. Manually calculate the maximum possible execution times of these sequences. Have the Processor's real-time counter interrupt and begin the sequences sufficiently in advance of the computation starting times that nonsimultaneity of input/output and computation is preserved. Input/Output completion timekeeping can be accomplished by a table in the Local Scheduler, by an interrupt from the I/O Controller, or both.

A possible interleaving of computations and input/output actions for the QUANTIZED scheme is shown in Figure 7. For the mix of iteration rates shown, the Processor interrupt rate (by the real-time counter) is a little more than twice the highest iteration rate of "A", the highest-rate computation

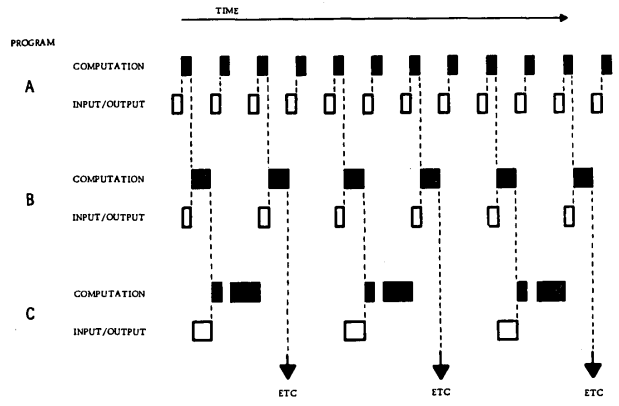

Figure 7 – A practical schedule for the QUANTIZED scheme

(By introducing still more uniformity into the time allocations for input/output sequences, a simpler pattern, more programming flexibility, and still fewer interrupts are achieved.) Several implications of Figure 7 are worth noting:

a. Input/Output and computation never occur simultaneously for computations of any given iteration rate. However, input/output for one iteration rate may occur simultaneously with computations of a different iteration rate.
b. The transport lag through the multiprocessor system (Controller input to Processor computation to Controller output) for computations of any one iteration rate is equal to the inverse of that iteration rate. This is illustrated in the detailed view shown in Figure 8. However, for a system of interlocking computations of different iteration rates, some care must be exercised to avoid transport lag uncertainties and to provide transport lags characteristic of the highest iteration rates rather than of the lowest iteration rates.

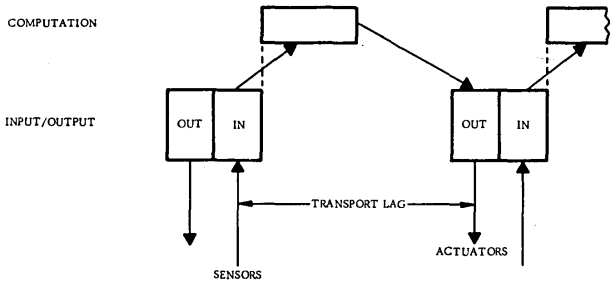

Figure 8 – Detail view of the schedule of Figure 7

Our experience has shown the QUANTIZED scheme to be of greater interest than the NESTED scheme. The QUANTIZED scheme has the further advantage of being achievable with a conventional Processor because the High Rate and Low Rate Registers are not required.

CONCLUSIONS

This paper has presented a periodic program scheduler and two schemes for controlling I/O traffic in real-time computation systems. These schemes have been shown to be very efficient in terms of use of processor time while providing the following important features:

1. Guaranteed precise scheduling of periodic programs
2. No requirement for individual programmers to set up linkages to other programs

3. Call of I/O variables just prior to program execution to limit processor waits, to provide fresh sensor data, and to prevent simultaneous computation and input/output.
4. Easy adaptation to changes in the periodic program sequences because of either the need for reconfiguration or changes in computation load

The real advantages of the complete scheduler are realized when a heavy load of precisely periodic programs must be processed. This situation is known by the authors to occur in the newer central avionics and future space computation systems; however, it may well occur in a number of other real-time control systems. In these heavily loaded systems, not only is efficiency important but also a large number of program changes generally occur in development and in the field. The Local Scheduler is easily able to adapt to these changes without loss of scheduling efficiency. As a result, the complete scheduler appears to have broad potential for application to real-time control systems.