CAS
PL-94-028 Lodz ul. Chodkiewicza 1a POLAND
techsupp@cas.com.pl; www.cas.com.pl
tel/fax: 48' 42 686 25 47; 48' 42 686 50 28

IMPLEMENTATION OF MONITOR CONCEPT IN MODULA-2

Abstract

This paper describes an implementation of the monitor concept as a couple of Modula-2 modules which provide the possibility to use interrupts together with structural synchronisation primitives on the application layer of the real time programs. The proposed implementation of monitor concept has been successfully used in several real time systems as a low level operating platform to build onto. One of them is software for the real time flight inspection system of radio-navigation aids illustrates the scope of possible application of the proposed monitor concept implementation.

INTRODUCTION

Number of the real-time applications increased during the last decade and this tendency will be likely to continue in the nearest future. Unfortunately, the main progress has been made rather in the hardware than in the software, thus new practical approaches to programming methods are still needed. Software designers can receive two kinds of support. First, it is a provision of:

- New methods of solving particular issues,
- Languages and programming tools (i.e. more efficient and friendly compilers, linkers, editors, and, what is most important, the new generation of debuggers), and first of all,
- Good programming principles.

Another kind of support can be received from system libraries or operating systems, modules of which can be used as an active part of the whole program. This paper is a contribution to the latter area. A set of MODULA-2 modules providing implementation of the monitor concept for the real-time applications is considered.

There is no consensus on the definition of the real-time programming. It is only obvious that *time* is a notion which must be included into consideration. But how to do it is not so obvious. Analysing the needs from the software designer point of view it has to be stated that the time has many aspects, and can be treated:

- *As a constraint imposed on the program execution.* Wirth expresses this as necessity of taking the execution speed of utilized processors into account during the program validation process.
- *As the driver of a mechanism to schedule particular actions*. This aspect appears when some actions of a computer system have to be performed on schedule, i.e. synchronously with the astronomical time.
- *As the driver of a mechanism to share the processor*. Usually each real-time system consists of some concurrent activities. Therefore there must be a way to switch a processor (or processors) between them. One of the possibilities is to transfer the control after expiration of a time limit.
- As *a measure of speed of some external or internal processes*. This aspect appears when a system has to react to the frequency of incoming events.
- *As a measure of the correctness of behavior of the system* constituent parts. Comparing time intervals between particular events with constraints imposed on the program and technological process under control, frequently it is possible to recognize some failures, and consequently react to them. This enables to treat time as a program correctness guard and to design system with the stepwise degradation.

Second, a very important feature of the real-time systems is their intrinsic *reliability*. Reliability is crucial because failure of the computer system could result in an economic disaster or even loss of human lives. Unfortunately, there are no appropriate means to guarantee full reliability, but it can be significantly increased by using well-designed programming tools, programming methods and by complying with the principles of the structured programming during the program life cycle. We need such tools and methods, which give us the possibility to verify a program. It is much desired that as many things as possible could be already checked at an early stage of the program development process. With respect to the system reliability the programming language and multiprogramming mechanisms is regarded as the most influencing ones the program development process. Our earlier examination of the problem and gained experiences lead us to choose Modula-2 as a programming language, and the monitor concept as a method for management and co-operation of concurrent processes. We find these solutions as the most safe and simple to use and implement.

The third distinguishing feature of the real-time systems is that an *environment* under which a computer operates is an active component of the whole system. This implies the need to design the program in such a way that it could co-operate with external devices performing parallel partial tasks belonging to and constituting to the technical process under control. Some of these devices may be other computers. To overcome that, the way to exchange data between the main processor and the environment, and to synchronise external events with particular operations of the program must be provided. The exchange of data issue can be simply solved by using memory-mapped I/O. Synchronisation of external

events with the program can be achieved by using hardware interrupts. An interrupt is an electrical signal which reports to the processor that a certain, important for control and monitoring the technical process behaviour event, coupled with it, has just occurred, and therefore some appropriate actions should be initiated to handle it. The event is formally represented by a condition. It can be observed that the interrupt concept is the same as the signal concept proposed by Hoare, and should result in a similar syntax of operations relating to both of them.

It should be noted that the use of interrupts in the real-time systems is not only the privilege of a "system" part of the program, but they must also be accessible higher in an application software layer. Obviously, increase of reliability can be obtained by the provision of appropriate structural mechanisms, what is the reason to introduce the interrupt notion into a whole concurrent programming concept as its consistent part.

The monitor concept is a well-known structural approach to maintain co-operation among concurrent processes. It is especially useful when a uniprocessor system is used. Many implementations of the monitor concept are known, but most of them are assumed to be used for operating systems of general-purpose computers. For the mentioned solutions, only the Modula, language was designed to be used for the real-time applications. Unfortunately, Modula does not provide any facilities referring to the time utilisation, and the proposed interrupt handling mechanisms lead to necessity of relaxation of the original scheduling principle. This language suffers also from many impractical solutions.

This paper shows some details of an implementation of the monitor concept as a couple of Modula-2 modules which provide the possibility to use interrupts, together with ordinary and time guarded synchronisation primitives by the application program. They offer also a time-sliced scheduler and some support to handle run-time errors.

The proposed implementation of monitor concept has been successfully used in many real time systems as a low level operating platform to build onto. One of them is software for the real time flight inspection system of radio-navigation aids (NAVAIDS). Very short description of this system illustrates the scope of possible applications.

IMPLEMENTATION OF THE MONITOR CONCEPT

In the Modula-2 language the **SYSTEM** module, which gathers almost all standard facilities for concurrency, uses the coroutine concept. Coroutines are like processes, but are executed quasi-concurrently, sharing one processor, which is switched from one coroutine to another by explicit transfer statements. The definition of this module may vary, but it was assumed that it exports at least the following, referring to concurrency procedures: *NEWPROCESS* (to create a new coroutine), *TRANSFER* (to transfer control from one coroutine to another) and *IOTRANSFER* (to release the processor until an interrupt occurs).

The mechanisms for multiprogramming provided by the **SYSTEM** module must be considered as low-level facilities. Their simplicity, an advantage with regard to flexibility, appears to be a disadvantage concerning verifiability, reliability and readability.

One of the main advantages following the use of Modula-2 is the modular expandability. Hence concerning flexibility the lack of the appropriate structural mechanisms supporting concurrency offered by the language can be regarded as an advantage, provided that there is an extension of the language - a module that provides structural operations designed to create, synchronise and communicate of concurrent processes, as well as all operations needed to manage programs behaviour in various situations. The proposed **MANAGER** module is such the extension. All multiprogramming mechanisms exported from the module have been designed under the assumption that for purpose of the process-to-process communication the well-known monitor concept is used.

Each concurrent process is an independent thread which contends with each other for resources such as processor, I/O devices, buffers, and so on. A process is created and initiated by the issue of the *PROCEDURE STARTPROCESS(P: PROC; WorkArea: CARDINAL; Priority: CARDINAL)* exported from the ***MANAGER*** module. *P* is a parameterless global procedure, *WorkArea* conveys the information about how much memory is needed for the created process. The work area is allocated from the heap by the parent process (the creating one). The *Priority* points out relative importance of the process among another processes. More detailed discussion on this subject is given later.

Processes scheduling principles.

If one processor is designed to perform some concurrent processes there must be a way to appreciate their relative importance to schedule them. Generally the scheduling policy is built around either the deadline or the priority notions. Because the hardware world uses exclusively the priority, in the presented implementation the basic scheduling mechanisms uses also this notion.

Three kinds of processes with regard to their urgency and, what follows, to their priority can be distinguished:

a) Interrupt driven,
b) Co-operating directly with the interrupt driven processes,

c) Others.

It is obvious that processes constituting the "a" group should pre-empt processes from other groups, and therefore should have the highest priority. Consistently, processes from "c" should have the least priority. Of course this simple rule cannot be applied without additional stipulations. For example, a process from group "c" while performing an inter-process synchronisation, and, what follows, some scheduling operations, must not be pre-empted. But this is equivalent to a temporally increase of its priority to the highest possible level. Hence, it is inevitable that when running, the process has to change its priority and its classification. It was assumed that there must be a way to separate places where processes can handle interrupts and co-operate with others processes belonging to group "b". Hardware interrupts are handled in the order imposed by their priorities. It means that the current process can be pre-empted by an interrupt driven process. It is not unlikely that there are some processes coupled with the same priority. Therefore an additional mechanism must be used. It is proposed that the relative priority is bestowed upon each new created process. The relative priority is constant during the process life cycle. Within a given relative priority the FIFO algorithm is applied to set up execution order.

One of the most important multiprogramming problems is to assure consistency of processed data. A way to solve this problem is to use the monitor concept. According to the definition, the monitor is a part of a program that consists of a set of procedures and its private, local data. The monitor allows only one process to be active (executing a procedure) within the monitor. Therefore the monitor's data consistency can be assured.

Concerning Modula-2, monitors can be implemented as modules with priorities specified at their headings. If a uni-processor system is applied the mutual exclusive access to common resources could be simply assured by disabling all interrupts when a process enters any monitor, but this solution is unacceptable for the real-time applications. To overcome this problem, the monitor priority is coupled with the interrupt priority in this way that the process after entering a monitor with a certain priority disables all the interrupts with equal and less priority. However, there must be fulfilled additional principles. First, the point where a process handling an interrupt is resumed, due to the interrupt occurrence, must lay inside a monitor, which has equal or greater priority than the interrupt. Second, processes within monitors with a certain priority must not call procedures from monitors with less priority. This principle is checked at run-time.

The monitor is an ideal construct to point out areas where a certain degree of the processes importance could be achieved. Therefore the monitor boundaries are the places where some scheduling decision should be taken. Because from the monitors with a priority the process can call only procedures from monitors with equal or higher priority it cannot decrease priority, and as a consequence there cannot appear any more important process than the currently executed one. Therefore when entering a next monitor the process should be always continued - no scheduling actions are needed. After leaving one monitor, the process may return to monitor with less priority or leave the last monitor. Then it should be pre-empted by another one, if any, which have higher priority. More precisely the following scheduling principles are proposed:

> *The process which reduces the main priority from η to λ, as a result of leaving a monitor with priority η, is keeping the processor unless there is a ready to run process within a monitor with priority π, where $\eta \geq \pi > \lambda$ or there is an interrupted process within the priority $\pi = \lambda$.*

This principle leads to a very easy algorithm. Obviously, it does also assure the mutually exclusive access to resources inside monitors, because the only way to pre-empt a process inside a monitor by another one is to accept an interrupt. But the interrupt driven process and all processes activated by it must have higher priority, and thus cannot access the monitor with the interrupted process inside.

Process to process communication.

In concurrent programming, there is the need to communicate processes with each other. This refers to both communication among internal processes with themselves, and internal with external ones. The aim is to synchronise certain operations and to exchange data. The monitor concept offers the signals as a way of the synchronisation and the monitor as the place where data can be safely exchanged. Each signal is associated with an important event (condition); hence the appearance of a signal is meant as appearance of the associated event, which manifests in establishing the relevant condition. Generally, three kinds of operations on signals are defined, namely, *wait* - to suspend process until the associated signal will be sent, *send* - to awake one of the waiting processes, and *awaited* - to check if any process is waiting for the specified signal.

After issuing the wait operation the current process releases all the monitors which it posses, thus it must leave all relevant data in a consistent state. There must be initiated a scheduling mechanism to choose another process to run, because the processor is released as well.

Like wait, the send signal operation needs some scheduling decisions. Two distinct policies are essentially possible. According to the first, after sending a signal the process should immediately awake the waiting one, if any, and give up

the monitor to it. When uniprocessor system is applied, it means that the process issuing the signal is suspended and the control is transferred to the signalled one. The suspended process will afterwards regain the processor. This kind of scheduling let treat the waiting process as a continuation of the process that has established the awaited condition. The main advantage of the above solution could be revealed when the program validity is proved because the monitor is not released at all, and thereby there is no possibility to change the data enclosed by the monitor, and the established condition as well, by another, third process.

The second policy for the scheduling scheme, implemented for example in Mesa, is based upon the principle that the signalling process keeps control, and the signalled one changes only its state and becomes ready to run. Of course, it cannot be as before assumed that the announced condition is still fulfilled when the signalled process is resumed, because other process, taking precedence, may have changed it in the meantime. Therefore, the signalled process, just after taking the control, should check again the condition, except that it cannot be changed, and, if necessary, wait once more for its occurrence.

Each of the discussed above scheduling disciplines relating signalling has advantages and disadvantages. However, they must be differently appreciated in various situations, because they are complementary to each other in a certain degree. It was decided, therefore, that both of them have been implemented. To avoid confusion, every time a signalling operation is used, there must be made the explicit distinction. To do it two kinds of signalling variables were defined: *Signal* and *Condition*, and, what follows, two signalling operations: *PROCEDURE SendSignal(S: Signal); PROCEDURE NotifyCondition(C: Condition);* and two wait operations *PROCEDURE WaitSignal(S: Signal);PROCEDURE WaitCondition(C: Condition);*

The described above set of operations can be applied to synchronise only internal processes. As it was stated, the synchronisation of internal and external processes by means of interrupts can and should be expressed in a similar way, except, what is obvious, signalling operations. According to Wirth, the wait for a signal incoming from outside (interrupt) can be written down as *DOIO(IntNum);* where IntNum is an interrupt identifier (customarily a cardinal number). Application of the signal concept and the *DOIO* operation allows implementation of the software processes synchronisation, and the synchronisation between software and hardware processes in the same way. As far as the scheduling discipline is considered the *DOIO* operation is similar to the ordinary wait operation, especially during its first phase when after calling the process is suspended until the denoted interrupt incomes. But one important difference must be observed during the second phase after the signal has been sent. Before issuing an internal signal the programmer can establish data consistency, and therefore all relevant monitors are ready to reschedule. Because each interrupt arises asynchronously, the program must be constructed in such a way that when an interrupt is accepted and the reschedule takes place the resumed process can gain access only to that monitors which are not occupied at the time by any process. The monitor priority concept, described above, overcomes this problem.

Time-sharing.

In the real-time applications usually almost all processes consume only short intervals of the processor time while gaining the control, because they are slowed down in a natural way by devices that are attached through the I/O channels. They release the processor by themselves. However, some processes, which use output devices like CRT monitors, LED displays or are purely numerical, can work continuously, i.e. do not have to wait for the end of any I/O operation or other events, and should usually work as fast as possible. Therefore, there must be a way to switch over occasionally the processor among them. One of the possible solutions is to call the *PROCEDURE* Release; exported from **MANAGER** that should be inserted into such processes to give up the control to the next ready to run process. The usefulness of this solution is limited and sometimes not sufficient because there is no acceptable method to choose the release points to slow down the processes in the same degree. The more adequate and comfortable solution is a time-sliced scheduler.

The time-sliced scheduler has been implemented in the **MANAGER** module. The reschedule takes place under the following assumptions:

- There is at least one ready to run process,
- The current process slice has expired, and
- The current process is not inside any monitor.

The time-sliced scheduler must be inevitably coupled with the hardware timer interrupt. But unfortunately, to implement any interrupt handler the **MANAGER's** services must be used. There are two reasons why handling this interrupt inside the module, which is to provide common purpose multiprogramming facilities, should be omitted. First, because the module consequently would have had to provide some time related operations, what unnecessarily increases its functionality and decreases flexibility of the whole solution. Second, because this would have distinguished one of the interrupts, what must be regarded as inconsistency and lack of orthogonality.

It is proposed to overcome the problem by exporting the procedure PROCEDURE StartTimeSlicing(IntNu: CARDINAL; RecoveryTime: CARDINAL); from the ***MANAGER*** module which permits to indicate one of the available interrupts (*IntNu*) to be used to measure the lapse of time by counting occurrences of the chosen one. It is assumed that this interrupt occurs and is handled elsewhere periodically. By this means the time notions was "literally" removed from the scheduling mechanism. The *RecoveryTime* parameter fixes how long processes can occupy monitors and when it expires the process is halted as erroneous.

Processes states.

Execution of the synchronisation operations results in changes of processes states. After a process has been created and activated by *STARTPROCESS* its state becomes *RUNNING* (Fig. 1). In the uniprocessor system there is only one running process at the same time. If a process wants to postpone the execution of subsequent operations until a condition will be established, and so it has issued the *WaitCondition*, *WaitSignal*, or *DOIO* operation, it becomes *BLOCKED*. Commonly, there are some ready to run processes. They are in the *READY* state waiting for their turn. The ready to run processes are connected with each other in an ordered queue in accordance with processes indexes. The index is a cardinal number which is calculated by the scheduler, which takes into consideration the value of the greatest module priority the process is currently inside, if the process has been interrupted, and its relative priority.

In other words, processes change their states as the result of appearance of some events or execution of the synchronisation operations. Execution of a synchronisation operation by one process can cause appearance of a certain event relevant to another process. For example, the issue of *NotifyCondition* by one process is perceived by another one as an event - condition appearance, and so it causes the change of the notified process state to *READY*; *WaitCondition, WaitSignal, DOIO* cause that the processor becomes disengaged what is the important event for the first ready to run process (Fig. 1), etc. The relationships between all possible events and their causes are drawn up in TABLE 1. Some of them will be discussed later.

The mentioned above three states are the main states which processes pass through. The *ERROR* state is reached when a run-time error is recognised. The remaining two states, namely, *ENDED, NON-EXISTENT* (Fig. 1) relate to the end of processes life time phase, and are a result of the acquired memory management strategy - each process has its own heap. If a process has reached its last statement in the procedure that constitutes its body, it is meant that its job has been just fulfilled, and that the process can be disposed, and so its state is changed to *ENDED*. To create a new process the parent process must allocate some memory from its own heap as the work area for its "child". Therefore, any process must not be disposed until all its descendants have been already disposed. After its last statement has been executed and all the termination procedures have been called it becomes *ENDED* and occasionally it is woken up for the purpose to dispose a next child. The process remains in this state as long as there is any child to dispose. After it has changed the state to *NON-EXISTENT*, which means that the memory it used is temporarily joined to a pool, and it will be further deallocated by the parent process. The whole program ends if there is no ready to run process and no process is waiting for an interrupt. Before the process state has been changed to *ENDED* it should release all gained resources. There are

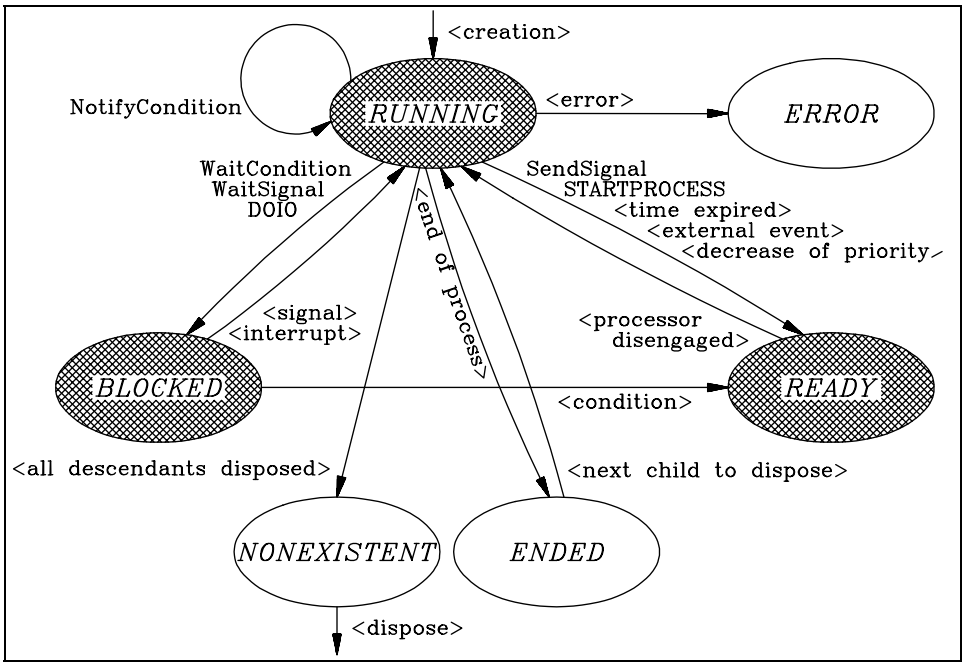

Fig. 1. Processes states.

two possibilities. The programmer can explicitly release them or they can be taken away.

The **MANAGER** module offers possibility to declare a set of procedures, which will be executed just after the end of the process. The list of procedures is ordered using stack algorithm - first declared is executed as last. But there are still possibilities to make mistake, therefore before the process work area would have been deallocated **MANAGER** checks if its heap is entirely empty, and otherwise the process is treated as erroneous.

Table 1. Events and their causes.

Cause	Event
SendSignal	<signal>
NotifyCondition	<condition>
STARTPROCESS	<creation>
<external event>	<interrupt>
<error>	<processor disengaged>
1<time expired>	<processor disengaged>
WaitCondition, WaitSignal, DOIO	<processor disengaged>
<all descendants disposed>	<processor disengaged>
<end of process>	<next child to dispose>
occurs when a parent becomes RUNNING and its child is NON-EXISTENT	<dispose>

FLIGHT INSPECTION SYSTEM OF RADIO NAVAIDS

The described above implementation of monitor concept has been successfully used as a low level operating platform to build onto software for the real time flight inspection system of radio-navigation aids (NAVAIDS) called CFIS.

CFIS is used for the airborne evaluation of accuracy and performance of ground navigation facilities. The system provides the capability to inspect the following aids: -Instrument Landing System (ILS), ILS associated approach markers (MKR), VHF omnidirectional range (VOR), Distance Measuring Equipment (DME), Non-Directional Beacon system (NDB), communication (VHF) and radar systems.

It is a modern, computerised system designed for the acquisition, recording, processing, analysis, display, and reporting of flight inspection data. It acquires various conditional signals from the avionics.

Main tasks of this system are as follows:

- Measurement and registration all signals vital for evaluation of navaid work correctness from avionics during flight inspection.
- Monitoring in real time chosen values to be used by a flight inspector,
- Processing of data to establish features of inspected navaid,
- Making reports of the inspection analysis,
- Data maintenance and archive for postpone stability analyze.

The main objective is to enable the flight inspector to check correctness of the emitted signals by inspected navaid and consequently to certificate the device as conforming to international standards.

Fig 2 illustrates an example procedure of glide path flight inspection (GS). This transmitter is only one part of the instrument landing system (ILS), which enables to safely land under low visibility condition. It is provided to navigate precisely and follow the appropriate path in the vertical direction.

The main objective of the mentioned above procedure is to check the glide path angle. During this procedure the inspection plane equipped with CFIS system keep moving on a line with speed about 250km/h and try to follow the glide path. In this time all signals necessary to assess device correctness are gathered from a receiver. This receiver is a part of measurement channel.

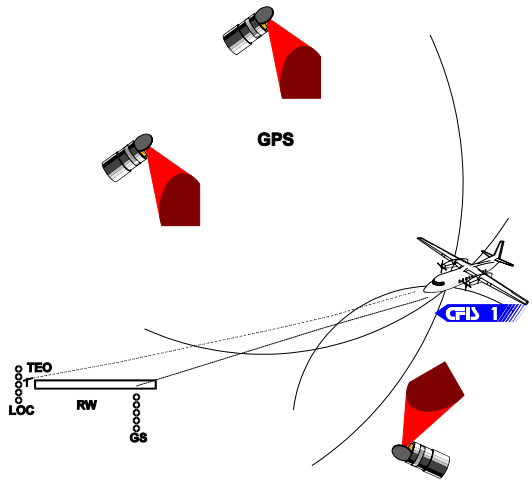

Fig 2: Flight inspection of ILS

RW - runway, TEO: radiotheodolit, GPS - Global Positioning System, LOC: localizer, GS: glide path