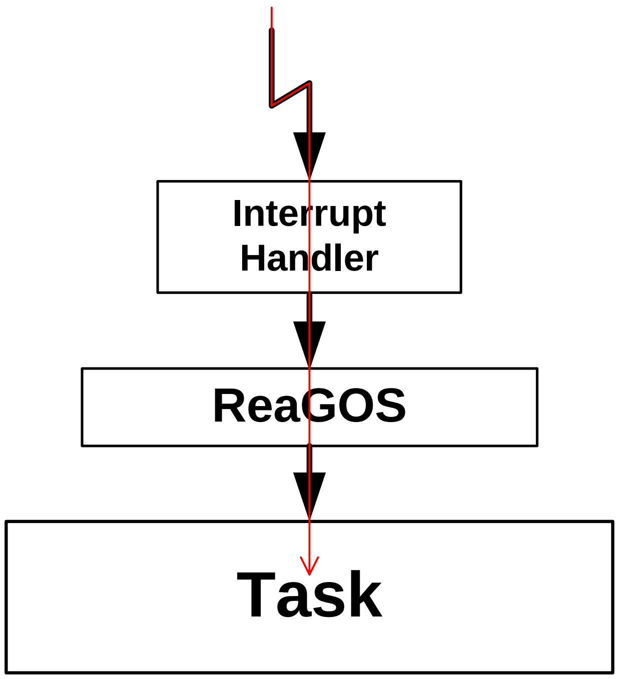

ReaGOS
- An Application Specific R/T-OS

Ari Okkonen, Mikko Levanto, Jyrki Okkonen, Antti Auer, Jarmo Kalaoja,
Heikki Päivike

Presented in the 5th Euromicro Workshop on Real-Time Systems, Oulu, Finland, June 22-24, 1993

ReaGOS - An Application Specific R/T-OS

Ari Okkonen, Mikko Levanto, Jyrki Okkonen, Antti Auer, Jarmo Kalaoja,
Heikki Päivike

Technical Research Centre of Finland (VTT),
Computer Technology Laboratory,
P.O.Box 201, SF-90571 Oulu, Finland,
E-mail: aok@tko.vtt.fi

Abstract

The real-time properties of most embedded systems are based on the use of real-time operating systems. Application of such operating systems requires high expertise and much time. The resulting software is often heavy and hard to maintain. In this paper we present a class of application specific operating systems called ReaGOS. ReaGOS is based on a new architecture and an operating principle, where the operating system calls application programs but not vice versa. The new architecture saves both data and code memory and it is fast enough for embedded systems. The operating system is generated automatically from a high level graphical specification.

1: Introduction

In embedded systems, the computer must respond to external stimuli with a reasonable speed although all previous stimuli have not been handled. The required response times of tasks depend on the nature of the tasks. For example, the control computer of a paper machine must give a status report while controlling the process. Often the external stimuli are statistically independent of each other. Therefore the computer cannot plan beforehand the order of event handling according to predicted arrival times of future events.

A computer always has a limited processing capacity. It is not always possible to complete handling of one event before handling of an urgent event must start. If an unlimited processor capacity were available, all event handling could be completed sequentially in the order of event arrival.

There are several different ways to solve the above problem. According to one method (nonpreemptive scheduling), all events are handled to completion one at a time. This method has some disadvantages: it cannot handle urgent tasks before handling of another event is finished. It is possible to avoid the problem by adding break points to the event handling code where the handling of urgent events can take place. On the other hand, the problem is not critical, if a powerful processor is available.

Another method uses interrupt mechanisms to handle I/O driver events [5]. This method has some disadvantages: when event handling is started it must be completed, even if remaining processing is less urgent than the interrupted process. This problem may be avoided by rapidly completing the event handling by producing the urgent responses and setting requests for continued handling of other responses. When handling of a previous event is completed, the system checks the requests and handles the non-urgent events. This kind of operating system may produce at most two responses of different priority from a single input event. An example of this is the Transputer [6]. The high priority may be used as interrupts and low priority may be used for batched event processing.

A third method is multitasking where several events can be handled simultaneously. The multitasking method may use different scheduling principles, such as priority, deadline scheduling, time slicing, or modifications of these basic methods. But there are also some problems when using the multitasking method: each task reserves part of working memory. The size of this memory area must be reserved according to the maximum usage which includes the memory space needed for the interrupt handling and operating system. Then the memory space for the operating system is reserved several times [8]. Tasks in multitasking method are endless loops which call the operating system to exchange data with environment. To test these programs, a genuine or simulated operating system is needed.

A fourth method is to start the tasks periodically. Then they may handle events independendly [7]. Each task has information which tells whether the task must be started or is still running. Usually the tasks using this method are not preemptive.

To conclude, the most operating systems for embedded systems are developed according to old principles having roots in the operating systems of large EDP service and time-sharing systems.

Theoretical foundations include scheduling theory and several theories of concurrent processes. Examples of good references are [1, 11].

2: Embedded Real-Time Operating Systems

According to our experiences in system development using different operating systems and in methodology research, we found that the use of available operating systems and concepts require:

- operating system specific tricks to implement the desired timing, communication and input-output operations.
- complex data and program structures to implement simple things.
- specific knowledge, not found in documentation, of how to use the operating system to meet the real time requirements.

Development time is wasted because:

- Mysterious errors emanate even if the manuals are followed to the letter.
- The applications are difficult to test.
- The errors are hard to spot.

Moreover, the final product is clumsy, because:

- The operating system is large because it contains so many unused features.
- The operating system is slow, because it is so complicated.

3: ReaGOS Approach

3.1: Goals

Our goal was to develop a concept with the following properties. It should:

- allow fast response times for some urgent events while heavy computation is under execution.
- support a data-flow based concurrent software architecture.
- support multiple waiting: any event concerning the task activates the task as soon as possible.
- support accurate timing without drift; it should be easy to implement precisely timed sequences.
- support easy implementation of inter-task data flows with user definable data types and static buffering.
- be based on static data structures for deterministic behaviour.
- support fast and small final system code; total code and data space of the system should be optimized.
- support straightforward and effective interrupt processing.
- include no excess operating system functions.
- be portable at the source code level.
- be suitable for automatic configuration and interfacing.

3.2: General Principles

We selected the following basic architecture for ReaGOS:

- preemptive scheduling with static priorities
- message based inter -task communication
- interrupts are handled directly by device specific interrupt handlers

Special features of ReaGOS are:

- It is generated specifically for the application.
- All tasks share the same stack.
- Operating system calls tasks, not vice versa.
- A task returns, if it wants to wait a new event. All output messages are launched on return.
- A task is called as soon as possible for any event concerning it, be it any message or time-out.
- Scheduling of lower priority tasks can be interrupted for a higher priority task.

3.3: Operation Principles

A task to be run under ReaGOS is actually a combination of a C-function (subroutine/procedure) and a block of data, *status block* [9]. The status block contains the state of the task, when the task is waiting for a new event. When an event that requires response from the task occurs, ReaGOS marks the event in the status block of the task and calls the function. When the task gets the response ready, it marks the response and the new state to the status block and returns.

ReaGOS consists of a scheduler, timer routine, a set of interrupt communication routines, and an initializing routine. The scheduler handles the events to and from the status blocks and calls the task functions and output drivers. The scheduler is called when it is possible that a task of higher priority needs to be activated.

The timer routine is called from a hardware timer interrupt handler. It requests a new call from the timer interrupt handler if needed for a new time-out and calls the scheduler for activation of the task that timed-out.

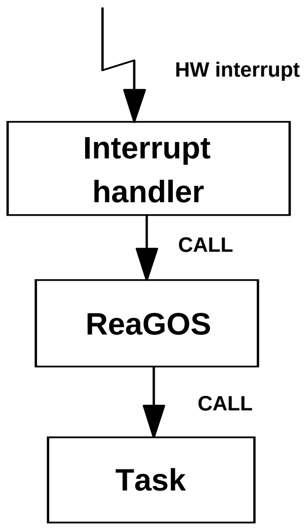

Fig. 1: The operating principle of ReaGOS.

An interrupt communication routine is called from a device interrupt handler to send data or a signal from the interrupt handler to a task. The interrupt communication routine manages data buffering and calls the scheduler if it is immediately possible to activate the receiving task.

In ordinary real-time operating systems each task needs its own stack because

1) When the task is interrupted the state of the task is contained in the stack.
2) Any task may be running while the others are suspended.
3) A running task needs a contiguous stack area with free space above the current top.

In our approach the tasks are ordered according to their priorities. The only reason that a task has state information in the stack while not running is that it is interrupted by a higher priority task or an interrupt service routine. If a task is waiting for a message or a time-out, all its state information is in a statically allocated status block. Moreover the interrupted task cannot be started before the processing of higher priority tasks and interrupt services are completed. When processing of a higher priority task or interrupt service is completed, the task space it used is released and it is free to be used by the task which is resumed.

The necessary preconditions of this arrangement are that a task does not call the operating system for any waits and there are no time-sharing-like task switching.

In our approach the tasks return when they start to wait for new events. When the event happens the ReaGOS calls the task function. Therfore, a task does not require any stack space during a wait .

3.4: Input-Output

ReaGOS does not include I/O services, but it gives facilities to implement effective device driver subsystems.

ReaGOS supports two level layered I/O architecture. The hardware level I/O is a combination of interrupt handlers and output handlers. The logical level I/O is embedded into tasks implementing the application.

Combinations of interrupt handlers and output handlers are called drivers.

An interrupt transfers the execution control directly to an interrupt handler routine. After completing the interrupt service, the handler returns directly from interrupt, if no messages need to be sent to tasks. Otherwise the handler calls a flow specific interrupt communication routine. An input interrupt handler may store data from several interrupts to the flow buffer before calling the interrupt communication routine. This reduces the overhead associated with internal communication and scheduling in block-oriented data transfer.

If data or a signal is sent from a task to an output handler, ReaGOS calls the output handler with interrupts disabled. On block-oriented output the transfer is initiated by an output handler and subsequent items are transferred by an accompanying output interrupt handler. The interrupt handler may in turn acknowledge completed transmission by sending a signal to a task. The buffering and flow control of output data must be handled explicitly by the I/O software.

A task communicates with a device driver in similar way as with another task. This eases testing in the host environment and makes applications easy to port to other hardware environments.

Interrupt handlers are written in terms of the hardware. Some C compilers allow the writing of interrupt handlers in C language. Otherwise some assembly language programming is needed.

An output handler is an ordinary subprogram. In most cases it can be written in C.

3.5: Timer Services

A task sees the timer services as two variables. One, `time_since_previous_call`, tells the task the duration of time since the previous activation of the task. With another, `max_time_to_next_call`, the task states the maximum duration of the next wait. Both durations define differences of nominal points of time when the task should have been or would be activated.

Using only nominal activation times, and counting their differences, inhibit drift (other than that of the H/W clock) in long or indefinite timing sequences.

It is the responsibility of the task to keep a sum of the elapsed time and maintain its internal timers.

These facilities allow any number of internal timers for a task. On entry each timer is decremented by the `time_since_previous_call`. A zero result means a time-out, which the task can handle. On exit the lowest value of the active timers is assigned to `max_time_to_next_call`. This information is sufficient for the operating system to activate the task on the next timeout.

In order to implement the timer services, ReaGOS needs a clock driver. The clock driver is device specific, not included in ReaGOS.

When ReaGOS needs the absolute time, it asks it from the clock driver. When ReaGOS wants a timer interrupt, it requests it from the clock driver. The timer interrupt request cancels any pending one. The ReaGOS may also cancel the request without giving a new one. When the requested timer interrupt occurs, the clock driver calls the appropriate entry of the ReaGOS.

ReaGOS requests the timer interrupt only when a timer activation is requested for a task with higher priority than the task currently in execution. Other timer requests could not cause task activation as long as the current task is under execution.

With suitable hardware this arrangement makes it possible to get timer interrupts only when task switching is needed, and so combine low overhead with accurate timing. This is an improvement to the traditional approach of getting interrupts at regular intervals.

3.6: Inter-Task Communications

Inter task communication is based on data flow. Any number of data flows can be used to connect two tasks or a task and a device driver. A data flow is specialized for one (C-language) data type. A special case of a data flow is a signal with no data content.

There are two basically different implementations of inter-task communication. Data flow from a lower priority task to a higher priority task or to an output driver is implemented by a shared store. Data flow from a higher priority task or from an interrupt driver is buffered.

There cannot be any shared storage problems from lower priority to higher priority because when the sender has stored the data to the shared store and returned, the higher priority task receives the data and keeps the processor busy until the message has been processed. There is no way for the lower priority task to disturb the store while the higher priority task is processing it. The higher priority process in return is not interested in the storage before the lower priority task has completed its transmission and returned. The drivers are executed with interrupts disabled and have the highest priority.

There are two kinds of buffering schema to be selected by the designer. In the static linkage schemata, both the transmitting task and the receiving task have static memory locations for the data. During transmission the data is copied from the static location to a ring buffer and upon reception the data is copied from the ring buffer to the static location. In the dynamic linkage schema the data is stored directly in the next free slot of the data flow buffer, and a pointer to the next data item in the ring buffer is given to the receiver. There is a trade-off between pointer copying and data copying. The decision is left to the designer.

The source code of ReaGOS is generated from a graphical task diagram using the ReaGeniX Priorizer [10]. The diagram specifies the tasks and their priorities, drivers, flows between them and, the buffering capacities

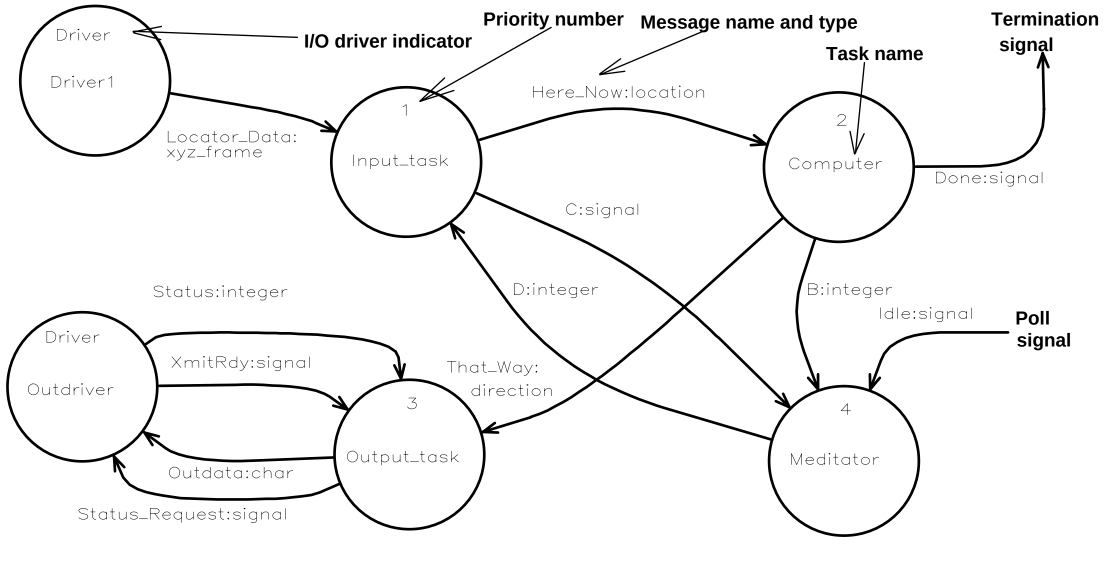

Fig. 2: An example of task allocation diagram.

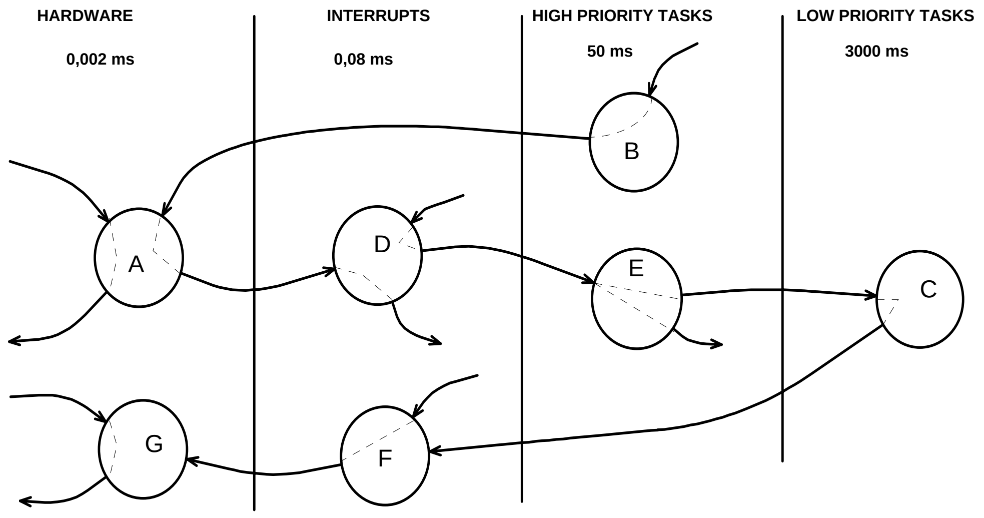

Fig. 3: Task Architecture Design

of the flows. ReaGOS code is compiled and linked with the application tasks. ReaGOS interfaces directly to the application module code generated by the ReaGeniX code generator.Hence, it is possible to specify the whole system using high level graphical language and implement it automatically using code generators.

In our approach, the task architecture is designed according to the external response time requirements [3]. This is possible because the tasks are always ready for any event, including time-outs, and there can be any number of concurrent subprocesses within a task. This is due to the coding style which gives rules for how to implement a state automaton and how to pack several concurrent automata to act as a more complex automaton. The ReaGeniX code generator implements these rules and releases the designer from the tricks of implementing concurrency, time-outs, and alternate triggering conditions. In traditional approaches it is very common to have several tasks to circumvent the mismatch between the specified functions and the capabilities of the operating system.

3.7: Properties of ReaGOS

The size of ReaGOS varies according to the number of the tasks and the number of flows. A ReaGOS generated for a system consisting of 12 data flows, 2 drivers, and 5 tasks took 4 kbytes of program memory. In addition to data space needed for the task state information and the buffers for the data flows ReaGOS needs some 10 bytes for each priority level for internal housekeeping. A contiguous stack space must be reserved so that the largest stack allocation of each priority level including interrupt drivers can fit simultaneously to the stack space. Note: if there are several tasks in the same priority level, only one of them can allocate stack at a time.

In a 386-PC with 20 MHz clock we have measured following times:

- Activation of a task from an interrupt handler via a data flow: 115 µs.
- Activation of a task on hardware time-out: 135 µs.
- Continuation of interrupt disable time, when an interrupt communication routine is called: 25 µs.
- Activation of a task from a lower priority task via a data flow: 100 µs.
- Activation of a task from a higher priority task via a data flow: 180 µs.

ReaGOS has been tested in IBM PC compatible hardware.

ReaGOS is written in ANSI-C using separate macro definitions. The main work in porting a system to other hardware will be rewriting of the device drivers, real-time clock driver, and the hardware setup routine.

A couple of demonstrations have been made using ReaGOS. The principles of ReaGOS were used to create a proprietary operating system used in mobile phones [2]. These examples have shown us that the original goals have been achieved.

4: Conclusions

We have found that the original goals are fulfilled with our current implementation of ReaGOS. ReaGOS seems to be suitable for real-life embedded systems with strict requirements.

Implementation of time-continuous flows could be improved to handle buffer overflows more regularly (even now it is not a disaster). An automatic interfacing to a traditional polling style background task would make it easy to use certain commercial tools to generate an user interface to the system.

A patent application is filed for the operating principle of the ReaGOS operating system [4].

5: Acknowledgments

The technology presented in this paper is developed in several R&D projects financed by VTT, the Technology Development Centre of Finland, and Finnish companies. We are grateful to our colleagues at the Technical Research Centre of Finland, especially Pertti Seppänen for his contribution.

References

[1] Mok, A.K., Fundamental Design Problems of Distributed Systems for the Hard Real Time Environment. Cambridge, Mass., Lab. for Computer Science, MIT. 1983. 183p.

[2] Mukari, T.: An Example of a Simple Real-Time Operating System, *in Embedded and Real-Time Systems*. Publications of the Finnish Artificial Intelligence Society - No 7, ISBN 951-96190-8-9. Helsinki 1992. (in Finnish)

[3] Okkonen, A. et al.: SOKRATES-SA — A Formal Method for Specifying Real-Time Systems. in Microprocessing and Microprogramming 27 (1989), pp. 1-5.

[4] Patent application FI 910479. 1991.

[5] U.S.Pat. 4,636,944

[6] U.S.Pat. 4,794,526

[7] U.S.Pat. 4,482,962

[8] European pat. 0 360 897 A1

[9] ReaGOS User's Manual. Technical Research Centre of Finland. Oulu 1992.

[10] ReaGeniX User's Manual. Technical Research Centre of Finland. Oulu 1992.

[11] Stankovic, J.A., Misconceptions about Real Time Computing - A Serious Problem for Next Generation Systems. IEEE Computer Society , October 1988. pp10-19.