RTDF: A REAL-TIME FORTH SYSTEM INCLUDING MULTITASKING

H.E.R. WIJNANDS and P.M. BRUIJN

Control Engineering Laboratory
Department of Electrical Egineering
Delft University of Technology
2600 GA Delft, The Netherlands

Abstract: This paper outlines a real-time Forth system, named RTDF. RTDF is intended for use as a development tool for single processor control systems. Due to the general language concepts applied, it has also been proved useful for discrete system simulation and other concurrent programming needs.

RTDF offers multiple task declaration, initiation and priority assignment. Synchronization and communication between tasks can be performed by the use of semaphores and monitors. Real-time facilities include the general use of timers and delay statements. An application running on a Z80 at 4 MHz may utilize a 5 ms sample time, maintaining an efficiency better than 95% for any number of tasks in any state.

Keywords: Process control, Discrete system simulation, Real-time programming, Concurrency, Multitasking.

1. INTRODUCTION

Microcomputers are increasingly being used to implement algorithms for control system applications. Most general purpose control systems employ the standard PID algorithm. To investigate and realize more complex controllers, the control engineer needs an interactive and flexible tool for real-time programming. Such a real-time system requires:

- multitasking facilities

  The program may be structured as a number of processes, tasks, which themselves are sequential, but are executed parallel. To communicate orderly the tasks should be able to share mutual excluded data. A method must be provided to synchronize tasks.

- a determined time response to events.

A task should be able to react immediately to events, such as the reaching of a specific time or the signaling of some other task or I/O port. The program may specify at which time (absolute) an action starts, or may specify a delay (relative) before an action takes place. In both cases a hardware provision should be available to measure time.

Due to the many advantageous properties Forth has been chosen as implementation environment; special attention has been given to the efficiency and portability of the resulting code.

2. MULTITASKING FACILITIES

The real-time Forth system RTDF supports multitasking. To simulate parallel processes on a one processor system, the execution of task s instructions is interleaved with the instructions of all other tasks. This simulation is completely transparent to users: there are no special requirements for the use of stacks or program loops; a task may be structured just like any other definition.

TASK DECLARATION AND INITIATION

The syntax for the declaration of a task is:

:TASK *task.name* routine-text ;

Example: :TASK watchdog BEGIN noise @ UNTIL bark ;

To initiate the task, the name is simply written (e.g. watchdog). Tasks can be initiated from the console, source text or by compiled code. When the task name is encountered the task can be executed; at the same time the calling program can also continue to execute. Actual execution depends on its priority with respect to the priorities of other active tasks.

A program that references an executing task will continue to execute, but the referenced task is initiated only if it is not already in execution: tasks may not be used recursively. The scheduler will keep track of the number of references: if a task is referenced N times it will sequentially execute N times.

Under the condition of sufficient memory resource, any number of tasks can be defined or executed at any time.

Example:

```
:TASK  controller  ...  ;   ( define the tasks )
:TASK  operator    ...  ;
:TASK  display     ...  ;

controller                   ( start the tasks  )
operator
display
```

PRIORITIES

Priority can be assigned to a task by:

(priority) PRIORITY *task.name*-list

Example: 100 PRIORITY watchdog

Active tasks with equal priority are executed in parallel and active tasks with the highest priority are executed first.
The priority value (priority) is an integer in the range from 0 to 255. Tasks have a default priority of 10. The priority 0 has a special meaning to the scheduler: the corresponding task will be aborted if execution is attempted.

SEMAPHORES

Semaphores in RTDF are modeled after the semaphores in Algol 68.

Semaphores are declared before use:

SEMA *semaphore.name*

The only allowed and compiler secured operators on a semaphore are WAIT, SIGNAL, SETSEMA and READSEMA.

Usage:

```
                *semaphore.name*  WAIT
                *semaphore.name*  SIGNAL
   (semavalue)  *semaphore.name*  SETSEMA
                *semaphore.name*  READSEMA  (semavalue)
```

WAIT inspects the integer variable contained within the semaphore. If its value is positive it is decreased by one. If, on the other hand, it is zero the task is suspended temporarily until the variable becomes positive. It is then decreased by one and the task is awakened.

SIGNAL increments the value of the integer variable contained within the semaphore. SIGNAL can result in the awake-

ning of other temporarily suspended tasks.

The default value of the semaphore is one. The value can be altered by SETSEMA and read by READSEMA.

MONITORS

The main purpose of the monitors is to facilitate programming in a multitasking environment. A monitor in RTDF is a declared part of the program in which only one task may be active at run time. This task must leave the monitor before any other task is allowed to enter.

Declaration:

```
:MONITOR monitor.name program MONITOR;
```

where *program* can be any sequence of declarations (colon definitions, task definitions, etc.).

Example 1:

```
:MONITOR control.parameters
     0 VAL p , d , i
     : get.parameters p d i ;
     : put.parameters TO i TO d TO p ;
MONITOR;
```

The moment a task fetches the parameters p, d, and i, the security exists that none of these parameters will be changed by other tasks.

Example 2:

The following illustration of the use of monitors is a solution of a classical concurrent problem, "The Bounded Buffer Problem".

One task is producing characters one at a time while another task is consuming them one at a time. The communication is done through a buffer which can hold a fixed number of characters. The producer may append records into this buffer as long as it is not full. The consumer may remove records as long as the buffer is not empty.

A solution using monitors in the notation of its inventor Brinch Hansen in DP (1):

```
monitor X is
     BUFFER: array(0..9) of CHAR
     IN, OUT: INTEGER
```

```
procedure APPEND(E:in CHAR);
     when IN < OUT+10 : BUFFER(IN mod 10) := E;
     IN := IN+1; end;
end APPEND;

procedure REMOVE(E: out CHAR);
     when OUT<IN : E := BUFFER(OUT mod 10);
     OUT := OUT + 1; end;
end REMOVE

begin
     IN :=0; OUT := 0;
end X
```

A similar solution in RTDF is:

```
:MONITOR getin/out.mon
     0 VAL in , out
     0 VAR buffer 8 ALLOT
     : getin/out  in  out  ;
MONITOR;

:MONITOR append.mon
     : append
          BEGIN  getin/out  10 +  <  UNTIL
          buffer  in  10 MOD +  C!
          1 in +TO  ;
MONITOR;

:MONITOR remove.mon
     : remove
          BEGIN getin/out SWAP < UNTIL
          buffer  out 10 MOD +  C@
          1 out +TO  ;
MONITOR;
```

A process call to "append" will add a character (the one previous left on the data stack of the calling task) into the circular buffer if not full. A process call to "remove" will take a character out of the buffer if not empty (it leaves it on the stack of the calling task). Any process calling "append" or "remove" will be placed in queue if another process is already executing in the routine "append" or "remove".

3. REAL-TIME FACILITIES

DELAY STATEMENT

The delay statement can be used if the current task execution has to stop for a specified time.

Usage: (delay.time) DELAY

The delay statement will immediately place the task that executes this statement into a waiting state for the specified time.

TIMERS

For real-time use a delay statement can be used in most current languages only. In a class of applications this is unsatisfactory. It is often not the two instants of start of consecutive execution which will be separated by a period, but the end of one execution and the start of the next that is desired.

To facilitate programming of correct intervals or cycle times timers may be used.

A timer is a data structure declared by:

TIMER *timer.name*-list

Example: TIMER timer1 , timer2 , timer3

Only 5 operators are allowed on a timer:

(timer.value)	*timer.name*	SETTIMER	
	timer.name	WAITTIMER	
	timer.name	READTIMER	(timer.value)
	timer.name	STOPTIMER	
	timer.name	CONTTIMER	

SETTIMER will assign the value, (timer.value), left on the datastack, to the timer structure. While the calling task can continue to execute the value of the timer will be decremented every elapsed time unit until zero.

WAITTIMER will take the current value of the timer and will place the task executing this statement into a waiting state for a time interval that corresponds to this value.

READTIMER returns the current value of the timer, STOPTIMER stops the timer value to be decremented and CONTTIMER will continue the decrementing.

Under the condition of sufficient memory resource any number of timers may be active at any moment. The efficiency with numerous timers is high in RTDF since, in fact, only one timer is counting; all other timers have offsets to this one timer. Administration is only carried out at zero descend of this one timer.

Example:

```
TIMER timer.pid

: PID-controller
     BEGIN
          20  timer.pid  SETTIMER
          calculate
          stimulate
              timer.pid  WAITTIMER
     REPEAT ;
```

Although the routines "calculate" and "stimulate" may take several clock cycles, the cycle time of the controller (20) is secured.

4. SOME REMARKS ON REAL-TIME RESPONSE

High speed real-time applications rely heavily on the definitions for task communication. This communication can be performed by:

- sharing data

When tasks share data on the basis of monitored operators, a special mechanism is provided for the real-time requirements of tasks with high priority. This mechanism is needed for the following situation:

If task A processes within a monitor it can be postponed by an activated task B with a higher priority. Then when task B wants to enter the monitored region, it will be placed in queue for later execution and task A is allowed to process again. As soon as task A leaves the monitor task B is awakened and since task A has lower priority task A will be suspended.
When only two tasks are active no problems are encountered. But we can imagine a situation in which other tasks are running, task A is within the monitored region and B is waiting to enter the monitor. Without any special provisions all other active tasks will now postpone task B, even if B has the highest priority' Of course this is undesirable. To overcome this problem the scheduler will temporarily assign the priority of task B to task A at the point in time that task B wishes to enter the monitor. When A leaves the monitor its original priority is restored. In this way a task awaiting the completion of a task of lower priority will not wait any longer than its priority allows. This scheme is vital for control systems, where task B supplies the calculated controller signal and A changes the parameters, as in adaptive control.

- signaling by semaphores

  Signaling takes place by semaphores. When a task has to await a semaphore or monitor, it will be placed in a queue. Tasks awaiting a common entry are served in most languages on the basis of first in first out without their priorities considered, which seems to be unfair. In RTDF scheduling is done on the basis of priority.

  If a signal is imposed on a semaphore the scheduler will immediately consider the consequences. This means that if a task with a suitable priority was awaiting this semaphore the scheduler will, within microseconds, perform a task switch.

In RTDF the time needed to respond to events lies in the millisecond range. The unit of time in RTDF is the cycle time of a hardware-provided periodical interrupt and may be as small as 5 ms for a Z80-system running at 4 MHz. This yields an efficiency of better than 95% with any number of tasks in any state.

5. PROGRAM DEVELOPMENT

Interrelated concurrent processes are often nondeterministic. Correctly appearing concurrent programs can have unexpected erroneous results and often have hidden errors. Although the language concepts here presented may support correct concurrent programs, the debugging phase is far more evident than in the case of sequential programming.

The most important feature of Forth is its ability to develop interactively. It is very important to preserve this programming environment for concurrent program development as well.

A large number of commands can be used for development. Their use requiers in some cases some knowledge of the scheduler. Development tools include commands to display scheduler status, the state of tasks, timers etc.. Interrupts can be simulated and a task switch can be forced. These commands are for development only. Usage in application programs can easily lead to unpredictable results.

RTDF is a complete Forth system. In addition to the real-time facilities it includes a powerful floating-point package, a filing system integrated to the CP/M operating system. Sources can be made by CP/M editors. For fast development, a Forth editor is available with a subset of Word Star commands.

System development for embedded systems has been proved to be fast. Special attention was given to the turnaround times

during development. The switching between Forth, CP/M and editors takes seconds. Forth code is compiled in record time. The compiling time of a complete new system, including its kernel takes less than 5 minutes.

6. CURRENT STATE AND FUTURE PLANS

While the first version is being used in a number of applications, further developments include a multiprocessor system for the control of a mobile robot and an expert system for process control implemented on a VME-bus system.

7. DISCUSSION

The language concepts presented by RTDF will help to develop correct concurrent programs. Their definitions are typically Forth-like and therefore include all the merits and demerits embodied by this language.

Although Forth is commonly used in real-time control systems there are no established Forth-tools in this field. (This is no surprise since concurrency is still one of the topics of todays computer language discussions.) The definitions here presented for real-time and multitasking applications may perhaps contribute toward a discussion on a structured development of these tools.

Reference:

(1) Brinch Hansen, "Distributed Processes", Commun. Ass. Comput. Mach., vol 21, Oct 1974.

--0--