

A Cheap Way To Monitor Task Status

Vincent Moscaritolo
Datavox Corporation
9 Pickering Way
Tancook Crescent, Suite 3A
Salem, MA 01970

Ever hear a bank talk?

RSX can be a very effective solution to the types of multi-tasking problems that occur in typical process control applications. I would like to describe one way in which I used some of the RSX parent-offspring tasking directives in an application called Banktalk.

Banktalk is a voice response system which uses the DECTalk DTC03 voice terminals to provide home banking functions. A customer can access account data (eg. savings balance), rate information, and perform account transactions by calling from a Touch-Tone phone.

A typical system can serve anywhere from one to 64 phone lines. Each line is managed by its own DECTalk and its own phone server task. Each server task runs under a unique terminal line. There is also a monitor task which tracks the status of each of the possible 64 server tasks.

I needed a simple way for each server task to communicate its current status (i.e., is it speaking, waiting for a call, crashed, etc.) to the monitor task.

The Server Task Table

I create a table in the monitor task that maintains information about each task.

```
;      Server Task Table Entry Format
;
;      +-----+
;      00 | Status          | 2 bytes
;      +-----+
;      02 | Taskname in R50 | 4 bytes
;      +-----+

T$STAT = 0           ; Offset to task status
T$ID = T$STAT + 2    ; Offset to task name
T$LEN = T$ID + 4     ; Length of a table entry
T$MAX = 64.         ; Maximum entries in table

TTABLE: .BLKB <T$MAX * T$LEN> ; The Server Task table
```

The table format is simple. Each server task maintains an entry in the table. The entry describes the task name and the current status.

Given this information, what I need now is some way to fill in the status part of the corresponding task's entry. The problem is each task is in its own data space and can't access the server task table.

Emitting Status

RSX provides several ways to communicate between tasks; I wanted a cheap one. Fortunately, the CNCT\$ and EMST\$ directives provide us with one such way. The CNCT\$ directive allows us to make a one way, one time connection between tasks. In fact, this connection is closed whenever the connected task either exits or emits status. The same mechanism that returns exit status can be invoked at any time. In the case of our server task, all it has to do is:

```
EMST$$      #status
```

If things are set up right, "status" can be picked up at the task that initiated the connection. Incidentally, "status" is preferably some number greater than 4, since 0 through 4 might be confused as one of RSX's reserved task exit status codes.

0	EX\$WAR	Warning; task may have succeeded
1	EX\$SUC	Success
2	EX\$ERR	Error
4	EX\$SEV	Severe error; system may be compromised

I don't know what happened to status 3, but the above info should be in section 4.2 of the Exec reference.

You might have noticed that in the above EMST\$ example I did not specify any first parameter. This informs RSX to pass status to any task connected to this one. You can also pick a specific task to receive status by placing the task ID here.

Connecting It Up

So how does this status word get to the connector task? Let's walk through an example. Assume two tasks A and B. Task A executes a CNCT\$ to task B. This causes RSX to create and queue an Offspring Control Block (OCB) to task B.

Some time later, task B emits status. RSX fills the OCB with whatever task B specified via the EMST\$, and returns the OCB to task A. Now the OCB is accessible to task A. Thus a simple method of intertask communications is completed.

The connection between A and B, however, is lost. In order to do it again, task A must respecify the connection to task B (assuming that B is still hanging around, of course). One way of doing this is

through the AST option in the CNCT\$ directive. We can set up a piece of AST-driven code that reconnects to whatever task emits status to it.

Enough talk, let's see some code.

```

;      Test program

NAME:   .RAD50   /  FIDO/

START:  MOV      #TTABLE, R3                ; Point at table
        MOV      NAME, T$ID(R3)           ; Load taskname
in
        MOV      NAME+2, T$ID+2(R3)       ; table entry
        CALL     STASK                    ; Start task up
        ... code continues ..

;      Start Task Subroutine
;
;      Inputs:   R3 - Points to task table entry
;
;      Outputs:  R0 - DSW status

SNAME:  .RAD50   /...SRV/                  ; Server taskname

STASK:  RPOI$S   #SNAME,,,,,,,,, T$ID(R3)  ; Start task up
        BCS      1$                          ; Handle failure

        CNCT$S   T$ID(R3),, #UPAST, T$STAT(R3) ; Make connection

1$:     MOV      $DSW, R0                    ; Get status
        RETURN

;      Table Update AST Service Routine
;
;      Inputs:   R3 - Points to task table entry
;
;      Outputs:  Table updated

UPAST:  MOV      R3, -(SP)                   ; Save R3
        MOV      2(SP), R3                  ; Point at entry
        CNCT$S   T$ID(R3),, #UPAST, T$STAT(R3) ; Reconnect task
        MOV      (SP)+, R3                  ; Restore R3
        TST      (SP)+                      ; Clear stack

        ASTX$S                               ; Exit from AST

```

"START" sets up a task name of "FIDO" in the first record of the Task Table. It also sets up R3 to point to that record. Now we can call STASK.

At STASK the RPOI\$ directive requests that a task be created from the prototype task at SNAME. In this case I call it "...SRV". It should be installed prior to the call, otherwise the directive punts and STASK returns IE.INS status in R0. I have elected to specify what task name "...SRV" clone will be called.

(Note that this option exists only on M-Plus and Micro/RSX. If this option is omitted, the created task's name is based on which terminal it runs on; e.g. if on TT10:, the task is called SRVT10.)

If all goes well with the RPOI\$, we proceed to connect to task FIDO. Note how we do the CNCT\$. We specify that an AST be queued when the connection is completed. We also set up T\$STAT(R3) to have FIDO's status written into. If all goes well, we should return to START with R0 set to IE.SUC. We can now go on our merry way, confident that T\$STAT(R3) contains the emitted status from FIDO.

Whenever FIDO emits status, UPAST is called. It should find a pointer to the OCB at (SP). Because ASTs can occur at almost any point in the program, I save R3 for later use, and loaded it with the top of stack. R3 now points to the appropriate task table entry, in this case FIDO's. Then I execute a CNCT\$ back to FIDO and set up the OCB at FIDO's record address. Next, I restore the state of things so I can exit from the AST.

Notice that it doesn't matter which task emits status, because I can tell which task it is by looking at the OCB address. (Which happens to also be the address of the appropriate server task table entry.)

When Connecting is Not Enough

Now that we have created what amounts to a reusable link with the EMST\$ / CNCT\$ pair, how can we improve on it? Let's try a few ideas I have used in my application.

Running on Other Terminals

In the above example, the prototype task "...SRV" is set up to be cloned a number of times. However, they are all running on the same terminal line. It would be nice if there were some way to also specify to the created task which terminal it should attach to. We can solve this problem by using one of those many parameters in the RPOI\$ that we previously ignored.

The RPOI\$ directive allows us to specify which terminal the created task runs under. This is done with the "dname" and "unit" parameters, where "dname" is the ASCII device name of the terminal (typically "TT") and "unit" is the unit number. By using this parameter, the "...SRV" task can be written as if it were doing I/O to its standard TI:.. There are three caveats, however.

Note that in order to run a task on terminal other than one's own, the task executing the RPOI\$ has to be Taskbuilt or, at the very least, installed privileged. In this case I specify a /PR:0 option in my TKB command.

A task can be created only on a physical device. For instance, you can use "TT" for a device name, but it doesn't really work for a pseudo device.

If the task is running on a terminal that isn't logged in (most of the time in my application), there is no User Block for the task. This causes all sorts of problems if you use named directories since RSX tries to use your UIC as a default directory name. Just remember if you open any files to specify the full filename.

Passing a Command Line

With RPOI\$ we can also pass a command line on startup to the cloned task. We use the "bufadr" and "buflen" option, where "bufadr" is the address of a string to be passed to the task and "buflen" is the length in bytes. This command line can be picked up by the target task by executing a GCML\$ directive. This is useful for passing filenames and options to the server tasks.

Timestamping Status

It is sometimes useful to know when was the last time a task emitted status. We can get this information by doing a GTIM\$ in the AST. A logical place to put it would be in the appropriate task table entry. Then, the next time we check a task's status, we can also find out how old this information is.

Display Update Flag

If the task table is used to provide information for a real time display, it is very helpful for the redisplay algorithm to have a quick way of knowing if something has "recently" changed in an entry. A good way to do this is by having the AST set something in the task table entry whenever it runs. The display routine can then update the corresponding screen field and clear the table entry.

Use Your Imagination

I have talked enough about what I have done. The best way to find out about what RSX can do for you is to sit down, code it up and try it out for yourself. I include here some code which improves on the previous example and includes some of the ideas just mentioned. Happy Hacking.

```

;      Server Task Table Entry Format
;
;      +-----+
;      00 | Status          2 | T$Stat
;      +-----+
;      02 | Taskid (r50)    4 | T$id
;      +-----+
;      06 | TTn             2 | T$TTn
;      +-----+
;      10 | Command line    2 | T$Cmd (Null Terminated)
;      +-----+
;      12 | Update Flag     1 | T$Updat
;      +-----+
;      13 | Time HMS        3 | T$HTime, T$MTime, T$STime
;      +-----+

T$STAT = 0 ; Task status
T$ID   = T$STAT + 2 ; Task name in
R50
T$TTN  = T$ID + 4 ; Terminal Number
T$CMD  = T$TTN + 2 ; Command line
erg
T$UPDAT = T$CMD + 2 ; Update byte
T$TIME  = T$UPDAT + 1 ; Timestamp field
T$HTIME = T$TIME ; Hours byte
T$MTIME = T$HTIME + 1 ; Minutes byte

```

```

        T$STIME = T$MTIME + 1           ; Seconds byte
        T$LEN   = T$STIME + 1           ; Length of entry
        T$MAX   = 64.                   ; Maximum entries

TTABLE: .BLKB      <T$MAX * T$LEN>      ; Server task
table

SNAME:   .RAD50    /...SRV/             ; Server taskname

; Start task
;
; Inputs: R3 - Points to task table entry
;
; Outputs: R0 - DSW status

STASK:   MOV       R3, R0                ;
        ADD       #T$CMD, R0            ; Get cmd line
erg
        CLR       R1                    ; Zero line
length

; Find length of string by looking for terminating null

1$:      TSTB     (R0)+                  ; Is this a null?
        BEQ      2$                    ; If so, exit
loop

        INC      R1                    ; Bump string
Ingth
        BR       1$                    ; And go look
again

; Request offspring task, pass OCB, and connect to offspring

2$:      RPOI$    #SNAME,,,,,,T$CMD(R3),R1,,#"TT,T$TTN(R3),T$ID(R3)
        MOV      $DSW, R0              ; Get status
        BCS     3$                    ; On failure go
die

        CNCT$S   T$ID(R3),, #UPAST, T$STAT(R3) ; Connect to task
        MOV      $DSW, R0              ; Get status
        BCS     3$                    ; On failure go
d~e

; Task started OK. Save status about it in the task table.

        MOV      #5, TSSTAT(R3)        ; Set status of 5
        GTIM$S   #TIMBUF               ; Get system time

        MOVB     SEC, T$STIME(R3)      ; Set up start
time
        MOVB     MIN, T$MTIME(R3)     ; in table entry

```

```

        MOV      HOUR, T$HTIME(R3)           ;
        CLRB     T$UPDAT(R3)                ; Clear update
flag
3$:     RETURN                                ; Return to
caller

;      Table Update AST Service Routine
;
;      Inputs: R3 - Points to task table entry
;
;      Outputs: None

UPAST:  MOV      R3, -(SP)                   ; Save R3 on
stack
        MOV      2(SP), R3                   ; Task entry pntr
        CNCT$$S  T$ID(R3),, #UPAST, TSSTAT(R3) ; Reconnect to
task
        GTIM$$S  #TIMBUF                     ; Get system time
        MOV      SEC, T$STIME(R3)           ; Set up start
time
        MOV      MIN, T$MTIME(R3)           ; in task entry
        MOV      HOUR, T$HTIME(R3)         ;
        CLRB     TSUPDAT(R3)                ; Clear update
fiag
        MOV      (SP)+, R3                   ; Restore R3
        TST      (SP)+                       ; Clean off stack

        ASTX$$S                               ; Exit AST state

```