SUN/134.3

Starlink Project
Starlink User Note 134.3

B D Kelly[1]
A J Chipperfield

30 March 1992

ADAM
Guide to Writing Instrumentation Tasks
Programmer's Manual

[1]Royal Observatory Edinburgh

Abstract

This guide shows you how to write programs for use as part of instrumentation systems under ADAM Version 2. If you are a programmer experienced with ADAM Version 1, then refer to Appendix C which summarises some of the differences between the new ADAM tasking model and the old one.

Contents

1 Introduction **1**

2 A simple picture of a task **1**

3 Waiting for a command **2**

4 Carrying out a simple command **2**

5 A task with two actions **3**

6 Completion of a timed interval **4**

7 Completion of data input **6**

8 Kicking one action from another **7**

9 Receipt of messages from other tasks **8**

10 Cancelling actions **9**

11 Behaviour of the parameter system **11**

12 Compiling and linking **12**

13 Further reading **12**

A Making tea and coffee **13**
A.1 Making tea 13
A.2 Making coffee 15
A.3 Controlling tea and coffee making 16

B Cancelling in multi-task subsystems **21**
B.1 Controlling task 21
B.2 Controlled task 22

C For “old” programmers **25**
C.1 Introduction to the changes 25
C.2 Linking old-style tasks 25
C.3 NEEDS list checking 25
C.4 DTASK_RPON and DTASK_RPOFF 25
C.5 DTASK_ASTSIGNAL and DTASK_TSTINTFLG 26
C.6 TASK_ASTMSG (NAME, LENGTH, VALUE, STATUS) 26
C.7 Multiple calls to TASK_ASTSIGNAL, TASK_ASTMSG 26
C.8 AST enabling and disabling 26
C.9 Message reschedules 26
C.10 Timeout on AST and message reschedules 26
C.11 Closing down the Parameter system – Global associations 26

D List of TASK routines **27**

E REQUEST constants 33

F Important STATUS values 33

G Document changes 34

1 Introduction

An ADAM instrumentation system typically consists of a number of separate programs which are loaded into various computers and which carry out their functions in response to receiving commands. A large part of their functionality involves either sending commands to some instrument and receiving data back from it, or sending commands to other ADAM tasks. It follows that communication is a key feature of an ADAM task, and one can expect that a task spends most of its time waiting for a communication of one sort or another. Experience indicates that it is very inconvenient if a task is only sensitive to the communication it is expecting - for example, it becomes very difficult for the user to intervene because there has been a change of plan. An ADAM task written as part of an instrumentation system should, therefore, be organised such that if it is waiting for something it can also receive a command. This document describes the ADAM facilities provided to enable you to write tasks which match this idea.

2 A simple picture of a task

A task is implemented by the application code plus an interface file. The interface file is a text file containing declarations of the parameters and actions (*i.e.* commands) which the task can carry out. The source code looks like this

```
SUBROUTINE MYTASK ( STATUS )
INCLUDE 'SAE_PAR'
INTEGER STATUS

IF ( STATUS .NE. SAI__OK ) RETURN
  do things
END
```

That is, it appears as a subroutine which is linked into some ADAM code called the task *fixed-part*. It is conventional to test STATUS on entry and to exit without action if it is not equal to SAI__OK. The value SAI__OK is defined in the include file SAE_PAR. When the task does things, it might have to wait for some other task or system to reply.

The things a task might be waiting for include

- receipt of a command
- completion of a timed interval
- completion of data input
- receipt of messages from one or more other tasks

An ADAM instrumentation application works by telling the ADAM system what kind of thing it is expecting to happen by giving it a REQUEST through calling TASK_PUT_REQUEST.

```
CALL TASK_PUT_REQUEST(REQUEST,STATUS)
```

and then returning to the code which called it (the task's fixed-part). See Appendix E for possible values of REQUEST.

The fixed-part waits for messages to arrive. If one arrives which your application has warned it about, then it will call your application after storing information about the message where you can collect it.

The following sections describe how to wait for the various possibilities.

3 Waiting for a command

When you first load your task, the only thing it is waiting for is a command. It can receive commands from other ADAM tasks telling it to SET or GET the value of one of its program parameters. The fixed-part handles this without your code being aware of it. It can also receive a command telling it to CANCEL an earlier command, but as there hasn't yet been a command, the fixed-part will return an error to the other task.

If the fixed-part receives an OBEY <action_name> command, it checks whether the given action_name has been declared in the interface file. If it has, then your code is called. You can find out why your code has been called by using the TASK library, for example,

```
CALL TASK_GET_NAME(NAME,STATUS)
```

returns the name of the action.

4 Carrying out a simple command

Suppose your application is very simple, for example, SUMS.IFL contains

```
interface SUMS
   parameter VALUE
      type '_REAL'
   endparameter
   action SQUARE
      obey
      endobey
   endaction
endinterface
```

and SUMS.FOR contains

```
      SUBROUTINE SUMS ( STATUS )
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INCLUDE 'ACT_ERR'
      INTEGER STATUS
      REAL VALUE
```

```
            IF ( STATUS .NE. SAI__OK ) RETURN
            CALL PAR_GETOR ( 'VALUE', VALUE, STATUS )
            CALL MSG_SETR ( 'ANS', VALUE**2 )
            CALL MSG_OUT ( ' ', 'answer is = ^ANS', STATUS )
            END
```

This can be built as a task called SUMS and loaded using ICL. When it is loaded, the fixed-part reads the interface file and discovers that the task has just one action, called SQUARE, and one parameter called VALUE. If you send a command to it of the form

```
      ICL> send sums obey square
```

then the task will prompt you for VALUE. It will attempt to square your reply and then return to the fixed-part. If you send the command again, you will not be prompted for VALUE. The task will remember the value of VALUE unless your application calls PAR_CANCL, or you KILL the task from ICL.

5 A task with two actions

Let us make the task SUMS able to accept two different commands. SUMS.IFL contains

```
      interface SUMS
         parameter VALUE
            type '_REAL'
         endparameter
         action SQUARE
            obey needs VALUE
            endobey
         endaction
         action ADD
            obey needs VALUE
            endobey
         endaction
      endinterface
```

and SUMS.FOR contains

```
            SUBROUTINE SUMS ( STATUS )
            IMPLICIT NONE
            INCLUDE 'SAE_PAR'
            INCLUDE 'ACT_ERR'
            INTEGER STATUS
            REAL VALUE
            CHARACTER*(PAR__SZNAM) NAME

            IF ( STATUS .NE. SAI__OK ) RETURN
            CALL TASK_GET_NAME ( NAME, STATUS )
            IF ( NAME .EQ. 'SQUARE' ) THEN
               CALL PAR_GETOR ( 'VALUE', VALUE, STATUS )
               CALL MSG_SETR ( 'ANS', VALUE**2 )
               CALL MSG_OUT ( ' ', 'answer is = ^ANS', STATUS )
            ELSE IF ( NAME .EQ. 'ADD' ) THEN
               CALL PAR_GETOR ( 'VALUE', VALUE, STATUS )
               CALL MSG_SETR ( 'ANS', VALUE+VALUE )
               CALL MSG_OUT ( ' ', 'answer is = ^ANS', STATUS )
            ENDIF
            END
```

As above, you can issue the ICL command

```
ICL> send sums obey square
```

and you will be prompted for VALUE. Thereafter, that same value of VALUE will be used, even if you

```
ICL> send sums obey add
```

However, notice that the interface file has now been changed to specify that VALUE is NEEDed by the actions. This means that you can

```
ICL> send sums obey add 42
```

and the value 42 is put into the parameter by the fixed-part before your application is called.

6 Completion of a timed interval

Having received a command, the application might not wish to complete immediately, but may want to wait for something to happen. The simplest thing to wait for is the passing of a timed interval. This is set up by using TASK_PUT_REQUEST(ACT__WAIT,STATUS) to tell the fixed-part a timer is required and using TASK_PUT_DELAY to tell the system how long a time is required. Note that your code has to find out whether it is being called for the first time or not by inquiring the sequence number SEQ.

```
interface timer
   action WAIT
      obey
      endobey
   endaction
endinterface

      SUBROUTINE TIMER ( STATUS )
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INCLUDE 'ACT_ERR'
      INTEGER STATUS
      INTEGER SEQ
```

```
            IF ( STATUS .NE. SAI__OK ) RETURN
            CALL TASK_GET_SEQ ( SEQ, STATUS )
            IF ( SEQ .EQ. 0 ) THEN
      *      first-time, request 100 millisecond wait
               CALL TASK_PUT_DELAY ( 100, STATUS )
               CALL TASK_PUT_REQUEST ( ACT__WAIT, STATUS )
            ELSE
      *      next time in, finished
               CALL MSG_OUT ( ' ', 'finished', STATUS )
            ENDIF
            END
```

The time delay can also be used as a timeout facility in conjunction with waiting for input or message receipt.

It is possible to have several actions active at once, each waiting to be called by the fixed-part. Here is a simple example of two actions, each doing a timed reschedule.

```
            SUBROUTINE TIMER ( STATUS )
            IMPLICIT NONE
            INCLUDE 'SAE_PAR'
            INCLUDE 'ACT_ERR'
            INTEGER STATUS
            INTEGER SEQ
            CHARACTER*(PAR__SZNAM) NAME

            IF ( STATUS .NE. SAI__OK ) RETURN
            CALL TASK_GET_SEQ ( SEQ, STATUS )
            CALL TASK_GET_NAME ( NAME, STATUS )
            IF ( SEQ .EQ. 0 ) THEN
               IF ( NAME .EQ. 'WAIT' ) THEN
                  CALL TASK_PUT_DELAY ( 5000, STATUS )
                  CALL TASK_PUT_REQUEST ( ACT__WAIT, STATUS )
               ELSE IF ( NAME .EQ. 'WAIT1' ) THEN
                  CALL TASK_PUT_DELAY ( 1000, STATUS )
                  CALL TASK_PUT_REQUEST ( ACT__WAIT, STATUS )
               ENDIF
            ELSE
               IF ( NAME .EQ. 'WAIT' ) THEN
                  CALL MSG_OUT ( ' ', 'WAIT has rescheduled', STATUS )
                  CALL TASK_PUT_DELAY ( 5000, STATUS )
                  CALL TASK_PUT_REQUEST ( ACT__WAIT, STATUS )
               ELSE IF ( NAME .EQ. 'WAIT1' ) THEN
                  CALL MSG_OUT ( ' ', 'WAIT1 has rescheduled', STATUS )
                  CALL TASK_PUT_DELAY ( 1000, STATUS )
                  CALL TASK_PUT_REQUEST ( ACT__WAIT, STATUS )
               ENDIF
            ENDIF
            END
```

Obviously, this application never terminates, but will put out the WAIT message every 5 seconds and the WAIT1 message every 1 second.

7 Completion of data input

Suppose that your task has to input data from some hardware connected to an RS232 line. You can use the VAX/VMS SYS$QIO system service with your own AST handler to start the input operation, optionally call TASK_PUT_DELAY to set a timeout on the operation, and return to the task fixed-part after having put a request ACT__ASTINT.

When the input completes, VMS hands control to your AST handler which can carry out whatever actions are necessary and then use TASK_ASTMSG(NAME,LENGTH,VALUE,STATUS) to tell the fixed part to call the application again. In this call:

NAME is the name of the relevant task action

VALUE is a character string containing any information you want passed

LENGTH is the number of significant bytes in VALUE

Provided NAME coincides with one of your actions which has requested to be called again, any associated timer is cancelled and your application is called.

TASK_GET_REASON(REASON,STATUS) returns, in REASON, value MESSYS__ASTINT if the message was received from the AST handler, or MESSYS__RESCHED if the timer completed. In the former case, you could use TASK_GET_VALUE(VALUE,STATUS) to get the information passed from the AST handler.

Let us consider an example task called DO_IO with an action called READ.

```
      SUBROUTINE DO_IO ( STATUS )
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INCLUDE 'ACT_ERR'
      INCLUDE 'MESSYS_ERR'
      INCLUDE 'DDMSG'
      INCLUDE '$IODEF'
      INTEGER ASTPARM
      CHARACTER*(MSG_VAL_LEN) VALUE
      INTEGER CHAN
      CHARACTER*80 INSTRING
      INTEGER REASON
      INTEGER STATUS
      INTEGER SEQ
      EXTERNAL READAST

      IF ( STATUS .NE. SAI__OK ) RETURN
      CALL TASK_GET_SEQ( SEQ, STATUS )
      IF ( SEQ .EQ. 0 ) THEN
         CALL SYS$ASSIGN ( 'TTA5:', CHAN, ,)
         CALL SYS$QIO ( , %VAL(CHAN), IO$_READ,, READAST, ASTPARM,
     :     %REF(INSTRING), %VAL(80),,,,)
         CALL TASK_PUT_DELAY ( 5000, STATUS )
         CALL TASK_PUT_REQUEST ( ACT__ASTINT, STATUS )
      ELSE
```

```
   CALL TASK_GET_REASON ( REASON, STATUS )
   IF ( REASON .EQ. MESSYS__ASTINT ) THEN
      CALL TASK_GET_VALUE ( VALUE, STATUS )
      CALL MSG_OUT ( ' ', VALUE, STATUS )
   ELSE IF ( REASON .EQ. MESSYS__RESCHED ) THEN
      CALL MSG_OUT ( ' ', 'timed-out', STATUS )
   ENDIF
   CALL SYS$DASSGN ( %VAL(CHAN) )
ENDIF
END

SUBROUTINE READAST ( ASTPARM )
IMPLICIT NONE
INCLUDE 'SAE_PAR'
INCLUDE 'DDMSG'
INTEGER ASTPARM
CHARACTER*(MSG_VAL_LEN) VALUE
CHARACTER*(PAR__SZNAM) NAME
INTEGER LENGTH
INTEGER STATUS

STATUS = SAI__OK
NAME = 'READ'
VALUE = 'input finished'
LENGTH = 14
CALL TASK_ASTMSG ( NAME, LENGTH, VALUE, STATUS )
END
```

8 Kicking one action from another

In some complex instrumentation systems you may wish to have an action in a task which wakes other waiting actions within the same task. TASK_KICK(NAME,LENGTH,VALUE,STATUS) may be called to do this.

```
SUBROUTINE KICKER ( STATUS )
IMPLICIT NONE
INCLUDE 'SAE_PAR'
INCLUDE 'ACT_ERR'
INCLUDE 'MESSYS_ERR'
INTEGER STATUS
INTEGER SEQ
INTEGER REASON
CHARACTER*(PAR__SZNAM) NAME
CHARACTER*(MSG_VAL_LEN) VALUE
INTEGER LENGTH

IF ( STATUS .NE. SAI__OK ) RETURN
CALL TASK_GET_NAME ( NAME, STATUS )
IF ( NAME .EQ. 'WAIT' ) THEN
   CALL TASK_GET_SEQ ( SEQ, STATUS )
```

```
      IF ( SEQ .EQ. 0 ) THEN
         CALL TASK_PUT_DELAY ( 10000, STATUS )
         CALL TASK_PUT_REQUEST ( ACT__WAIT, STATUS )
      ELSE
         CALL TASK_GET_REASON ( REASON, STATUS )
         IF ( REASON .EQ. MESSYS__KICK ) THEN
            CALL TASK_GET_VALUE ( VALUE, STATUS )
            CALL MSG_OUT ( ' ', VALUE, STATUS )
         ELSE IF ( REASON .EQ. MESSYS__RESCHED ) THEN
            CALL MSG_OUT ( ' ', waked by timer', STATUS )
         ENDIF
      ENDIF
   ELSE IF ( NAME .EQ. 'KICK' ) THEN
      VALUE = 'waked by kick'
      LENGTH = 13
      CALL TASK_KICK ( 'WAIT', LENGTH, VALUE, STATUS )
   ENDIF
   END
```

9 Receipt of messages from other tasks

Suppose that your task is controlling other tasks. Say it sends an OBEY message to some other task. The other task will carry out its operations, maybe generating output intended for the user to see, and send your task a final message when it has completed the obey. You can use:

```
TASK_ADD_MESSINFO(PATH,MESSID,STATUS)
```

to tell the fixed-part you are expecting messages on this PATH,MESSID combination, then return to the fixed-part having set a request ACT__MESSAGE. You could also have set a timeout as above. You can use multiple calls to TASK_ADD_MESSINFO if you are controlling more than one operation in other tasks.

When the fixed-part calls your application again, you can use:

```
TASK_GET_MESSINFO(PATH,CONTEXT,NAME,VALUE,MESSID,EVENT,STATUS)
```

to obtain the information carried in the message.

Note that any output generated by the other task (*e.g.* using MSG_OUT) will have been handled automatically by the fixed-part of your task and will not cause your application subroutine to be called. Messages from subsidiary tasks which do cause your subroutine to be called will generally be completion messages, however, a special 'TRIGGER' message is also available.

To send a TRIGGER message the subsidiary task can use:

```
TASK_TRIGGER(ACTNAME,VALUE,STATUS)
```

The message will be sent and the task can continue working.

When your controlling task calls TASK_GET_MESSINFO, MESSYS__TRIGGER will be returned in EVENT.

Obviously, the example for all this is rather complex, involving three separate tasks and multiple actions. The example, written by William Lupton, involves a pair of tasks for making tea and coffee respectively and a third task controlling them, and is relegated to Appendix A.

10 Cancelling actions

It is possible to cancel an action which is in a wait state. Here is a simple example.

```
interface timer
   action WAIT
      obey
      endobey
      cancel
      endcancel
   endaction
endinterface

      SUBROUTINE TIMER ( STATUS )
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INCLUDE 'ACT_ERR'
      INCLUDE 'ADAMDEFNS'
      INTEGER STATUS
      INTEGER SEQ
      INTEGER CONTEXT

      IF ( STATUS .NE. SAI__OK ) RETURN
      CALL TASK_GET_CONTEXT ( CONTEXT, STATUS )
      IF ( CONTEXT .EQ. OBEY ) THEN
         CALL TASK_GET_SEQ ( SEQ, STATUS )
         IF ( SEQ .EQ. 0 ) THEN
            CALL TASK_PUT_DELAY ( 10000, STATUS )
            CALL TASK_PUT_REQUEST ( ACT__WAIT, STATUS )
         ELSE
            CALL MSG_OUT ( ' ', 'finished', STATUS )
         ENDIF
      ELSE IF ( CONTEXT .EQ. CANCEL ) THEN
         CALL MSG_OUT ( ' ', 'I was cancelled', STATUS )
         CALL TASK_PUT_REQUEST ( ACT__CANCEL, STATUS )
      ENDIF
      END
```

Note the CANCEL declaration in the interface file. This example is exercised by

```
ICL> send timer obey wait
ICL> send timer cancel wait
```

It is possible to write the application such that CANCEL modifies the behaviour of the rescheduling action rather than terminating it.

```
interface timer
   parameter CANTIME
      type '_INTEGER'
   endparameter
   action WAIT
      obey
      endobey
      cancel needs CANTIME
      endcancel
   endaction
endinterface

      SUBROUTINE TIMER ( STATUS )
      IMPLICIT NONE
      INCLUDE 'SAE_PAR'
      INCLUDE 'ACT_ERR'
      INCLUDE 'ADAMDEFNS'
      INTEGER STATUS
      INTEGER SEQ
      INTEGER CONTEXT
      INTEGER STATE
      SAVE STATE, TIME

      IF ( STATUS .NE. SAI__OK ) RETURN
      CALL TASK_GET_CONTEXT ( CONTEXT, STATUS )
      IF ( CONTEXT .EQ. OBEY ) THEN
         CALL TASK_GET_SEQ ( SEQ, STATUS )
         IF ( SEQ .EQ. 0 ) THEN
            STATE = 0
            CALL TASK_PUT_DELAY ( 500, STATUS )
            CALL TASK_PUT_REQUEST ( ACT__WAIT, STATUS )
         ELSE
            IF ( STATE .EQ. 0 ) THEN
*            Normal reschedule
               CALL TASK_PUT_DELAY ( 500, STATUS )
               CALL MSG_OUT ( ' ', 'default timer', STATUS )
               CALL TASK_PUT_REQUEST ( ACT__WAIT, STATUS )
            ELSE
*            Rescheduling after CANCEL
               CALL TASK_PUT_DELAY ( TIME, STATUS )
               CALL MSG_OUT ( ' ', 'altered timer', STATUS )
               CALL TASK_PUT_REQUEST ( ACT__WAIT, STATUS )
            ENDIF
         ENDIF
      ELSE IF ( CONTEXT .EQ. CANCEL ) THEN
         STATE = 1
         CALL PAR_GET0I ( 'CANTIME', TIME, STATUS )
         CALL TASK_PUT_DELAY ( TIME, STATUS )
         CALL MSG_OUT ( ' ', 'timer changed', STATUS )
         CALL TASK_PUT_REQUEST ( ACT__WAIT, STATUS )
      ENDIF
      END
```

Then:

```
ICL> send timer obey wait
ICL> send timer cancel wait 10000
```

will cause the task to start rescheduling at 0.5sec intervals, but then switch to rescheduling at 10sec intervals. In this particular example the action never terminates.

An example of sending a CANCEL to a task which is controlling a subsidiary task is given in Appendix B.

11 Behaviour of the parameter system

As far as the ADAM parameter system is concerned, there are two aspects of instrumentation tasks which cause them to behave differently from standard data analysis tasks (A-tasks). The first is the way they are linked. The second is the form of the interface file declarations.

Whenever an A-task executes, the parameter system is started-up, the application is called, then the parameter system is closed-down. For an instrumentation task this continuous opening-closing of the parameter system does not happen. There are two obvious consequences of this. Firstly, parameters remain ACTIVE once they have been given a value. Secondly, global associations for WRITE never happen.

All the parameters of an instrumentation task are available to all the actions. In addition, a specific OBEY or CANCEL can specify that values of named parameters can be passed on the command line.

```
action TRYIT
   obey needs TIME
        needs MYVAL
   endobey
endaction
action OTHER
   obey needs MYVAL
        needs TIME
   endobey
endaction
```

The order in which NEEDS declarations occur specifies the order in which the values are expected on the command-line.

An item of special interest to programmers of instrumentation systems is the value-search-path (VPATH) declaration INTERNAL. An INTERNAL parameter has the following properties which distinguish it from ordinary parameters.

If it is a scalar, it is stored as a variable inside the parameter system. All other parameter values are stored in HDS. This means that access to INTERNAL scalar parameters is much faster than access to ordinary parameters. This property may be of interest to instrumentation applications which make heavy use of the parameter system.

An INTERNAL parameter is not prompted for. All other parameters have an implicit PROMPT at the end of their VPATH declarations unless it is explicitly overridden by NOPROMPT.

The behaviour of PAR_PUTxx is different for an INTERNAL parameter. Putting an INTERNAL parameter simply puts the given value into the parameter. For an ordinary parameter, the parameter system expects to be told an existing HDS component into which the value should be put.

12 Compiling and linking

Assuming that `$ ADAMSTART` has been obeyed to start ADAM, type:

```
$ ADAM_DEV
```

This will define logical names for include files such as SAE_PAR and ACT_ERR and display the message "`+ logged in for ADAM program development`". User-code for tasks may now be compiled.

To link instrumentation tasks, type:

```
$ ILINK task [qual] [*]
```

task (mandatory) is the list of user routines, libraries *etc.* required for input to the linker. Note that the ADAM libraries will be included automatically. The executable image produced will take its name from the first item specified.

qual (optional) is any qualifiers required for the linker (*e.g.* /DEBUG).

* (optional) specify `*` if it is required to link with ADAM object libraries rather than the shareable images. Note that the full ADAM release must be installed and `$ @ADAM_SYS:SYSDEV` obeyed for this option to work. If `*` is specified, `qual` must also be specified (use /NODEBUG if nothing else is required).

13 Further reading

More detailed descriptions of the subjects introduced in this document can be found in the following documents.

- SG/4 — ADAM – The Starlink Software Environment
- SG/5 — ICL – The Interactive Command Language for ADAM - Users Guide
- SUN/115 — ADAM – Interface Module Reference Manual
- SUN/104 — MSG and ERR – Message and Error Reporting System
- AED/15 — Using the ADAM Parameter System
- SUN/114 — PAR – ADAM Parameter Routines - Programmer's Guide (in preparation)

A Making tea and coffee

A.1 Making tea

```
interface test_tea
#  Test tea task
    parameter param
        type '_integer'
        range 1,10
        vpath 'internal'
        default 2
    endparameter
    action lapsang1
        obey
        endobey
    endaction
    action lapsang2
        obey
        endobey
    endaction
    action lapsang3
        obey
        endobey
    endaction
    action lapsang4
        obey
        endobey
    endaction
    action lapsang5
        obey
        endobey
    endaction
endinterface
```

```
      SUBROUTINE TEST_TEA (STATUS)

*     Test D-task that is run from a rescheduling control task

      IMPLICIT NONE
      INTEGER   STATUS          ! Modified STATUS

      INCLUDE   'SAE_PAR'
      INCLUDE   'ADAMDEFNS'
      INCLUDE   'ACT_ERR'

      INTEGER   SEQ             ! Action sequence number
      INTEGER   CONTEXT         ! Context (OBEY or CANCEL)
      INTEGER   PARAM           ! Arbitrary integer parameter
      INTEGER   DELAY           ! Delay between initial and final entries
      CHARACTER NAME*24         ! Action name
      CHARACTER VALUE*80        ! Value string
      SAVE PARAM                ! Value must be saved

      IF (STATUS .NE. SAI__OK) RETURN

*  Pick up required "ACT parameters"

      CALL TASK_GET_NAME (NAME,STATUS)
      CALL TASK_GET_CONTEXT (CONTEXT,STATUS)
      CALL TASK_GET_SEQ (SEQ,STATUS)

*  Loop through possible OBEYs

      IF (NAME(1:7) .EQ. 'LAPSANG') THEN
         IF (SEQ .EQ. 0) THEN
*         Produce error - undefined parameter on LAPSANG2
            IF (NAME(8:8) .EQ. '2') THEN
               CALL PAR_GET0I ('X',PARAM,STATUS)
               IF (STATUS .NE. SAI__OK) THEN
                  CALL ERR_REP (' ',
     :            'TEA: LAPSANG1 Deliberate error - '//
     :            'No parameter X: ^STATUS', STATUS)
*                Flush the error messages and allow task to continue
                  CALL ERR_FLUSH ( STATUS )
               ENDIF

*         Get parameter value on LAPSANG3
*         to be used as a count of TRIGGERS
            ELSE IF (NAME(8:8) .EQ. '3') THEN
               CALL PAR_GET0I ('PARAM',PARAM,STATUS)
               IF (STATUS .NE. SAI__OK) THEN
                  CALL ERR_REP (' ',
     :            'TEA: LAPSANG3 Failed to get PARAM: ^STATUS',
     :             STATUS)
*              Set harmless value for PARAM
                  PARAM = -1
                  RETURN
               ENDIF
            ENDIF
            CALL MSG_SETC ('NAME',NAME)
            CALL MSG_OUT (' ','TEA: Starting ^NAME action',STATUS)
            DELAY = 1000 * (ICHAR(NAME(8:8)) - ICHAR('0'))
            CALL TASK_PUT_DELAY (DELAY,STATUS)
            CALL TASK_PUT_REQUEST ( ACT__WAIT, STATUS )
         ELSE IF (NAME(8:8) .EQ. '4' .AND. SEQ .LE. PARAM) THEN
            VALUE = NAME(1:8)//' is paging you ...'
            CALL TASK_TRIGGER (NAME,VALUE,STATUS)
            IF (STATUS .NE. SAI__OK) THEN
               CALL ERR_REP (' ',
     :         'TEA: LAPSANG4 Failed to trigger control task: ^STATUS',
     :          STATUS)
            ENDIF
            CALL TASK_PUT_DELAY (2000,STATUS)
            CALL TASK_PUT_REQUEST ( ACT__WAIT, STATUS )
         ELSE
```

```
            CALL MSG_SETC ('NAME',NAME)
            CALL MSG_OUT (' ','TEA: Finishing ^NAME action',STATUS)
            CALL TASK_PUT_VALUE ('Lapsang''s ready!',STATUS)
         ENDIF

      ENDIF
      END
```

A.2 Making coffee

```
interface test_coffee
#  Test coffee task
    action mocha1
        obey
        endobey
    endaction
    action mocha2
        obey
        endobey
    endaction
    action mocha3
        obey
        endobey
    endaction
    action mocha4
        obey
        endobey
    endaction
    action mocha5
        obey
        endobey
    endaction
endinterface

      SUBROUTINE TEST_COFFEE (STATUS)

*     Test D-task that is run from a rescheduling control task

      IMPLICIT NONE
      INTEGER   STATUS          ! Modified STATUS

      INCLUDE   'SAE_PAR'
      INCLUDE   'ACT_ERR'

      INTEGER   SEQ             ! Action sequence number
      INTEGER   DELAY           ! Delay between initial and final entries
      CHARACTER NAME*24         ! Action name

      IF (STATUS .NE. SAI__OK) RETURN

*  Pick up required "ACT parameters"
```

```
      CALL TASK_GET_NAME (NAME,STATUS)
      CALL TASK_GET_SEQ (SEQ,STATUS)

*  Loop through possible OBEYs

      IF (NAME(1:5) .EQ. 'MOCHA') THEN
         IF (SEQ .EQ. 0) THEN
            CALL MSG_SETC ('NAME',NAME)
            CALL MSG_OUT (' ','COFFEE: Starting ^NAME action',STATUS)
            DELAY = 1000 * (ICHAR(NAME(6:6)) - ICHAR('0'))
            CALL TASK_PUT_DELAY (DELAY,STATUS)
            CALL TASK_PUT_REQUEST ( ACT__WAIT, STATUS )
         ELSE
            CALL MSG_SETC ('NAME',NAME)
            CALL MSG_OUT (' ','COFFEE: Finishing ^NAME action',STATUS)
            CALL TASK_PUT_VALUE ('Mocha''s ready!',STATUS)
         ENDIF
      ENDIF
      END
```

A.3 Controlling tea and coffee making

```
interface test_control
#  Test control task
    parameter max
        type '_integer'
        range 1,10
        prompt 'Number of actions'
        vpath 'prompt'
        default 1
    endparameter
    parameter time
        type '_integer'
        range -1,3600
        prompt 'Timeout in seconds'
        vpath 'prompt'
        default 10
    endparameter
    action brew
        obey
            needs max
            needs time
        endobey
    endaction
endinterface

      SUBROUTINE TEST_CONTROL (STATUS)

*     Test control task that controls and reschedules multiple actions
*     in multiple subsidiary tasks
```

```
      IMPLICIT NONE
      INTEGER   STATUS          ! Modified STATUS

      INCLUDE   'ADAMDEFNS'
      INCLUDE   'SAE_PAR'
      INCLUDE   'MESSYS_ERR'
      INCLUDE   'ACT_ERR'
      INCLUDE   'DTASK_ERR'

      INTEGER   I               ! Counter
      INTEGER   MAX             ! Number of TEA/COFFEE actions to start
      INTEGER   TIME            ! Timeout in seconds
      INTEGER   SEQ             ! Action sequence number
      INTEGER   PATH            ! Path to task in which OBEY completed
      INTEGER   MESSID          ! Message id of OBEY that completed
      INTEGER   TEA_PATH        ! Path to TEA task
      INTEGER   TEA_MESSID      ! Message ID of TEA's LAPSANG action
      INTEGER   TEA_ACTIVE      ! Number of active LAPSANG actions
      INTEGER   COFFEE_PATH     ! Path to COFFEE task
      INTEGER   COFFEE_MESSID   ! Message ID of COFFEE's MOCHA action
      INTEGER   COFFEE_ACTIVE   ! Number of active MOCHA actions
      INTEGER   CONTEXT         ! Context (OBEY or CANCEL)
      CHARACTER NAME*24         ! Action name
      CHARACTER VALUE*200       ! Action returned value
      CHARACTER INVAL*1         ! Action input value (unused)
      CHARACTER OUTVAL*1        ! Action output value (unused)
      INTEGER EVENT             ! Event which caused reschedule

      SAVE TEA_ACTIVE,COFFEE_ACTIVE,TIME ! Retain these values

      IF (STATUS .NE. SAI__OK) RETURN

*  Pick up required "ACT parameters"

      CALL TASK_GET_NAME (NAME,STATUS)
      CALL TASK_GET_SEQ (SEQ,STATUS)

*  Loop through possible OBEYs

      IF (NAME .EQ. 'BREW') THEN

*  First time through, initiate the actions ...

         IF (SEQ .EQ. 0) THEN

*  ... actions LAPSANG1 .. LAPSANG'MAX etc are initiated with a timeout
*  of TIME seconds ...

            CALL PAR_GET0I ('MAX',MAX,STATUS)
            CALL PAR_GET0I ('TIME',TIME,STATUS)

*  ... in the tea-maker ...

            TEA_ACTIVE = 0
            DO I = 1,MAX
               NAME = 'LAPSANG'//CHAR(48+I)
               INVAL = ' '
               CALL TASK_OBEY ('TEST_TEA',NAME,INVAL,
     :           OUTVAL,TEA_PATH,TEA_MESSID,STATUS)
               IF (STATUS .EQ. DTASK__ACTSTART) THEN
                  STATUS = SAI__OK
                  CALL TASK_ADD_MESSINFO (TEA_PATH,TEA_MESSID,
     :              STATUS)
                  TEA_ACTIVE = TEA_ACTIVE + 1
               ELSE
                  CALL MSG_SETC ('NAME',NAME)
                  CALL ERR_REP (' ',
     :              'CONTROL: Failed to start ^NAME: '//
     :              '^STATUS',STATUS)
*              Output reports associated with this failed OBEY
*              and try next
                  CALL ERR_FLUSH ( STATUS )
               ENDIF
            ENDDO

*  ... and in the coffee-maker ...

            COFFEE_ACTIVE = 0
            DO I = 1,MAX
               NAME = 'MOCHA'//CHAR(48+I)
               INVAL = ' '
               CALL TASK_OBEY ('TEST_COFFEE',NAME,INVAL,
     :           OUTVAL,COFFEE_PATH,COFFEE_MESSID,STATUS)
               IF (STATUS .EQ. DTASK__ACTSTART) THEN
                  STATUS = SAI__OK
                  CALL TASK_ADD_MESSINFO (COFFEE_PATH,COFFEE_MESSID,
     :              STATUS)
                  COFFEE_ACTIVE = COFFEE_ACTIVE + 1
               ELSE
                  CALL MSG_SETC ('NAME',NAME)
                  CALL ERR_REP (' ',
     :              'CONTROL: Failed to start ^NAME: '//
     :              '^STATUS',STATUS)
*              Output reports associated with this failed OBEY
*              and try next
                  CALL ERR_FLUSH ( STATUS )
               ENDIF
            ENDDO

*  ... and, if OK, set time-out period and set ACT__MESSAGE request.

            IF (TEA_ACTIVE .GT. 0 .OR. COFFEE_ACTIVE .GT. 0) THEN
               IF (TIME .NE. -1) THEN
                  CALL TASK_PUT_DELAY ( 1000*TIME, STATUS )
               ENDIF
               CALL TASK_PUT_REQUEST ( ACT__MESSAGE, STATUS )
            ENDIF
```

```
*  On subsequent entries, get the details of the message that has
*  caused this entry (it should either correspond to a subsidiary
*  action completion, TRIGGER or a timeout).

         ELSE
            CALL TASK_GET_MESSINFO (PATH,CONTEXT,NAME,VALUE,MESSID,
     :        EVENT,STATUS)

*  First check for timeout in which case abort the action ...

            IF (EVENT .EQ. MESSYS__RESCHED) THEN
               CALL MSG_SETI ('TIME',TIME)
               CALL ERR_REP (' ',
     :           'CONTROL: Timeout occurred after ^TIME '//
     :           'seconds',EVENT)

*  ... or check whether this is a triggering message, in which case
*  simply report and set ACT__MESSAGE request

            ELSE IF (EVENT .EQ. MESSYS__TRIGGER) THEN
               CALL MSG_SETC ('NAME',NAME)
               CALL MSG_SETC ('VALUE',VALUE)
               CALL MSG_OUT (' ',
     :           'CONTROL: Triggered by ^NAME: ^VALUE',
     :           STATUS)
               IF (TIME .NE. -1) THEN
                 CALL TASK_PUT_DELAY ( 1000*TIME, STATUS )
               ENDIF
               CALL TASK_PUT_REQUEST ( ACT__MESSAGE, STATUS )

*  ... or determine which action has completed. Set ACT__MESSAGE request
*  if more remain. Otherwise the BREW action is complete.

            ELSE
               IF (NAME(1:7) .EQ. 'LAPSANG') THEN
                  TEA_ACTIVE = TEA_ACTIVE - 1
               ELSE IF (NAME(1:5) .EQ. 'MOCHA') THEN
                  COFFEE_ACTIVE = COFFEE_ACTIVE - 1
               ENDIF

*  Report normal subsidiary action completion ...
               CALL MSG_SETC ('NAME',NAME)
               IF (EVENT .EQ. DTASK__ACTCOMPLETE) THEN
                  CALL MSG_OUT (' ',
     :            'CONTROL: Action ^NAME completed normally',STATUS)
*  including VALUE returned.
                  IF ( VALUE .NE. ' ' ) THEN
                     CALL MSG_SETC( 'VALUE', VALUE )
                     CALL MSG_OUT (' ',
     :               'CONTROL: Value string: ^VALUE', STATUS)
                  ENDIF
*  or failure ...
               ELSE
                  CALL ERR_REP (' ',
     :            'CONTROL: Action ^NAME completed: ^STATUS',EVENT)
*  including VALUE returned.
                  IF ( VALUE .NE. ' ' ) THEN
                     CALL MSG_SETC( 'VALUE', VALUE )
                     CALL ERR_REP (' ',
     :               'CONTROL: Value string: ^VALUE', EVENT)
                  ENDIF
*  Flush
                  CALL ERR_FLUSH ( EVENT )
               ENDIF
               IF (TEA_ACTIVE .GT. 0 .OR. COFFEE_ACTIVE .GT. 0) THEN
                  IF (TIME .NE. -1) THEN
                     CALL TASK_PUT_DELAY ( 1000*TIME, STATUS )
                  ENDIF
                  CALL TASK_PUT_REQUEST ( ACT__MESSAGE, STATUS )
               ENDIF
            ENDIF
         ENDIF
      ENDIF
      END
```

B Cancelling in multi-task subsystems

This example was provided by Ian Smith (ROE). When the controlling task receives a CANCEL, it sends a CANCEL to the subsidiary task. The controlling task then requests ACT__MESSAGE to wait for the final acknowledgement from the OBEY in the subsidiary task. The final completion status from the subsidiary task is obtained by the controlling task by the call to TASK_GET_REASON, irrespective of whether the final completion was due to a CANCEL.

B.1 Controlling task

```
interface cmotask
#  Motor task controller
   parameter motor
      type '_char'
      in 'FILTER_WHEEL', 'FOCUS_WHEEL'
      prompt 'which wheel - FILTER_WHEEL or FOCUS_WHEEL?'
   endparameter
   action motor_control
      obey
         needs motor
      endobey
      cancel
         needs motor
      endcancel
   endaction
endinterface

      SUBROUTINE CMOTASK( STATUS )

*     Control the task Motask which moves 2 imaginary motors

      IMPLICIT NONE              ! No implicit typing

      INCLUDE 'SAE_PAR'          ! Standard SAE constants
      INCLUDE 'ADAMDEFNS'
      INCLUDE 'MESSYS_ERR'
      INCLUDE 'ACT_ERR'
      INCLUDE 'DTASK_ERR'

      INTEGER STATUS             ! modified status
      INTEGER CONTEXT            ! context OBEY or CANCEL
      INTEGER SEQ                ! action sequence number
      INTEGER GOOD_STATUS        ! local status
      INTEGER MOTASK_PATH        ! path to subsidiary task
      INTEGER MOTASK_MESSID      ! message id
      CHARACTER*80 MOTOR         ! name of motor to be moved
      CHARACTER*80 INVAL         ! parameter string sent
      CHARACTER*80 OUTVAL        ! parameter string returned
      INTEGER REASON             ! subsidiary completion status

      IF ( STATUS .NE. SAI__OK ) RETURN
      GOOD_STATUS = SAI__OK
      CALL TASK_GET_CONTEXT ( CONTEXT, STATUS )
      IF ( CONTEXT .EQ. OBEY ) THEN
         CALL TASK_GET_SEQ ( SEQ, STATUS )
         IF ( SEQ .EQ. 0 ) THEN
            CALL PAR_GETOC ( 'MOTOR', MOTOR, STATUS )
            INVAL = ' '
            CALL TASK_OBEY ( 'MOTASK', MOTOR, INVAL, OUTVAL,
     :        MOTASK_PATH, MOTASK_MESSID, STATUS )
            IF ( STATUS .EQ. DTASK__ACTSTART ) THEN
               STATUS = SAI__OK
               CALL TASK_ADD_MESSINFO ( MOTASK_PATH, MOTASK_MESSID,
     :           STATUS )
               CALL TASK_PUT_REQUEST ( ACT__MESSAGE, STATUS )
            ELSE
               CALL MSG_SETC ( 'MOTOR', MOTOR )
               CALL ERR_REP ( ' ', '^MOTOR FAILED', STATUS )
            END IF
         ELSE
            CALL TASK_GET_REASON ( REASON, STATUS )
            IF ( REASON .EQ. DTASK__ACTCANCEL ) THEN
               CALL MSG_OUT ( ' ',
     :           'CMOTASK: subsidiary task has been cancelled',
     :           STATUS )
            ELSE IF ( REASON .NE. DTASK__ACTCOMPLETE ) THEN
               STATUS = REASON
               CALL ERR_REP ( ' ',
     :           'CMOTASK: subsidiary task has returned bad status',
     :           STATUS )
            END IF
         END IF
      ELSE IF ( CONTEXT .EQ. CANCEL ) THEN
         CALL TASK_CANCEL ( 'MOTASK', MOTOR, INVAL, OUTVAL, STATUS )
         IF ( STATUS .EQ. DTASK__ACTCANCEL ) THEN
            STATUS = SAI__OK
            CALL TASK_PUT_REQUEST ( ACT__MESSAGE, STATUS )
         ELSE
            CALL ERR_REP ( ' ', 'CMOTASK: failure cancelling MOTASK',
     :        STATUS )
         ENDIF
      END IF

      CALL PAR_CANCL ( 'MOTOR', GOOD_STATUS )

      END
```

B.2 Controlled task

```
interface motask
   parameter filter
      type '_integer'
      range 1,100
   endparameter
   parameter focus
      type '_integer'
      range 1,100
   endparameter
   action filter_wheel
      obey
         needs filter
      endobey
      cancel
      endcancel
   endaction
   action focus_wheel
      obey
         needs focus
      endobey
      cancel
      endcancel
   endaction
endinterface

      SUBROUTINE MOTASK( STATUS )

*     Task to drive 2 dummy motors. One called FILTER_WHEEL
*     and the other FOCUS. Each invocation requires 1 parameter
*     relating to requested position but will be used in the call to the
*     delay routine for a delay of n seconds

      IMPLICIT NONE              ! No implicit typing

      INCLUDE 'SAE_PAR'
      INCLUDE 'ACT_ERR'
      INCLUDE 'ADAMDEFNS'

      INTEGER STATUS
      INTEGER CONTEXT
      INTEGER SEQ
      INTEGER FILTER
      INTEGER FOCUS
      INTEGER PERIOD
      CHARACTER*(PAR__SZNAM) NAME

      SAVE FILTER
      SAVE FOCUS

      IF ( STATUS .NE. SAI__OK ) RETURN

      CALL TASK_GET_NAME ( NAME, STATUS )

      IF ( NAME .EQ. 'FILTER_WHEEL' ) THEN
         CALL TASK_GET_CONTEXT ( CONTEXT, STATUS )
         IF ( CONTEXT .EQ. OBEY ) THEN
            CALL TASK_GET_SEQ ( SEQ, STATUS )
            IF ( SEQ .EQ. 0 ) THEN
```

```
          CALL PAR_GET0I ( 'FILTER', FILTER, STATUS )
          PERIOD = FILTER * 1000
          CALL MSG_SETI ( 'FILTER', FILTER )
          CALL MSG_OUT ( ' ', 'moving to filter ^FILTER...',
:           STATUS )
          CALL TASK_PUT_DELAY ( PERIOD, STATUS )
          CALL PAR_CANCL ( 'FILTER', STATUS )
          CALL TASK_PUT_REQUEST ( ACT__WAIT, STATUS )
       ELSE
          CALL MSG_SETI ( 'FILTER', FILTER )
          CALL MSG_OUT( ' ', 'filter position ^FILTER reached',
:           STATUS )
       END IF
    ELSE IF ( CONTEXT .EQ. CANCEL ) THEN
       CALL TASK_PUT_REQUEST ( ACT__CANCEL, STATUS )
    END IF
 ELSE IF ( NAME .EQ. 'FOCUS_WHEEL' ) THEN
    CALL TASK_GET_CONTEXT ( CONTEXT, STATUS )
    IF ( CONTEXT .EQ. OBEY ) THEN
       CALL TASK_GET_SEQ ( SEQ, STATUS )
       IF ( SEQ .EQ. 0 ) THEN
          CALL PAR_GET0I ( 'FOCUS', FOCUS, STATUS )
          PERIOD = FOCUS * 1000
          CALL MSG_SETI ( 'FOCUS', FOCUS )
          CALL MSG_OUT ( ' ', 'moving to focus ^FOCUS...', STATUS )
          CALL TASK_PUT_DELAY ( PERIOD, STATUS )
          CALL PAR_CANCL ( 'FOCUS', STATUS )
          CALL TASK_PUT_REQUEST ( ACT__WAIT, STATUS )
       ELSE
          CALL MSG_SETI ( 'FOCUS', FOCUS )
          CALL MSG_OUT ( ' ', 'focus position ^FOCUS reached',
:           STATUS)
       END IF
    ELSE IF ( CONTEXT .EQ. CANCEL ) THEN
       CALL TASK_PUT_REQUEST ( ACT__CANCEL, STATUS )
    END IF
 END IF
 END
```

C For “old” programmers

C.1 Introduction to the changes

C-tasks, CD-tasks and D-tasks are now “unfashionable”, but are supported for the time being. That is, all existing ADAM tasks should continue to behave as they did under ADAM V1, except for the following two features.

Firstly, an OBEY to a task will no longer be rejected on the grounds that its NEEDS list is not satisfied. See section C.3 for more details.

Secondly, the VALUE string is only guaranteed to contain the parameter string for the OBEY on first entry to ACT. If the action is rescheduling, the VALUE string will not have retained the parameter string for subsequent entries.

Note that D-tasks are now free to use ERR and MSG and to send messages to other tasks whenever they feel like it (except inside AST routines!).

The characteristics of the new-style “instrumentation" tasks are as follows:

- The arguments previously passed in the call to CD-tasks and D-tasks, *ie.* CONTEXT, NAME, SEQ, VALUE and RETVAL are now accessed by calls to the TASK routines.
- The DTASK library is being phased-out as far as applications are concerned. The DTASK_ calls are replaced by TASK_ calls.
- The syntax of the interface file remains unchanged.

The following sections list all the changes.

C.2 Linking old-style tasks

Old-style D-tasks and CD-tasks may still be linked using DLINK and CDLINK. After re-linking they will use the new fixed-part and exhibit the new behaviour, particularly the improved error reporting. Furthermore, future changes to the fixed-part will be incorporated automatically via a shareable image.

DLINK and CDLINK have the same parameters as ILINK. Procedures DNOSHR and CDNOSHR are withdrawn.

C.3 NEEDS list checking

The fixed-part will no longer reject an OBEY or CANCEL on the grounds that the parameters specified on the NEEDS list do not have suitable values. The only significance of NEEDS lists in ADAM V2 is to specify the order of any command-line parameters – NEEDS constraints no longer have any effect.

C.4 DTASK_RPON and DTASK_RPOFF

These no longer do anything. The fixed-part automatically enables reporting back to whatever process issued the GET/SET/OBEY/CANCEL currently under way.

C.5 DTASK_ASTSIGNAL and DTASK_TSTINTFLG

These routines are being phased-out. TASK_ASTSIGNAL and TASK_TSTINTFLG, which have the same arguments and functionality, should be used instead. The DTASK_ calls will continue to work for the time being.

C.6 TASK_ASTMSG (NAME, LENGTH, VALUE, STATUS)

This is a new routine, similar in function to TASK_ASTSIGNAL, but allowing a message of length LENGTH to be passed in the character string VALUE. When the main-line code is rescheduled it can extract the message from VALUE.

C.7 Multiple calls to TASK_ASTSIGNAL, TASK_ASTMSG

It is now possible for an AST handler to make multiple calls to TASK_ASTSIGNAL and TASK_ASTMSG during a single execution. The information carried in the call is now sent as a message (rather than entered in a COMMON block) and so the last call does not overwrite the earlier ones.

C.8 AST enabling and disabling

The DTASK fixed-part no longer re-enables ASTs, and the various AST events (timed reschedules and AST reschedules) no longer disable ASTs. This means that execution of outstanding AST handlers is no longer delayed while the main-line code tidies-up from earlier events.

C.9 Message reschedules

The TASK library makes it possible for a task to send an OBEY to another task, and then request the fixed-part to reschedule this action on receipt of the final acknowledgement from the other task. The fixed-part automatically handles message forwarding from the other task.

C.10 Timeout on AST and message reschedules

If the application sets RETVAL and then returns with STATUS requesting an AST or message reschedule, the fixed-part will also set a timed reschedule for the time indicated in RETVAL. If the AST or message event happens first, the timer is cancelled. Implications of this are that if the application has been “accidentally" setting RETVAL in the past, it will now set up a timer, and alternatively, if a single action has been managing to set up multiple timers for itself, it will now only be able to have one outstanding at any one time.

C.11 Closing down the Parameter system – Global associations

The parameter system does not close-down when an instrumentation task action completes. This means that parameters retain their values and remain “active", rather than being returned to the “ground" state. *It also means that instrumentation tasks never write global associations.* This behaviour is similar to the behaviour for D-tasks, but unlike the behaviour for CD-tasks.

D List of TASK routines

The “action keyword” is the name by which a task’s action is known in the world outside that task. The “action name” is the name by which the action is known to the application code inside that task. These names are defined in the interface file. The action keyword defaults to the action name if it is not otherwise specified.

There is a similar distinction between parameter keywords and parameter names.

ACTKEY=CHARACTER*(PAR__SZNAM)	action keyword
ACTNAME=CHARACTER*(PAR__SZNAM)	action name
CONTEXT=INTEGER	symbol for GET, SET, OBEY or CANCEL
CONTEXTNAME=CHARACTER*(*)	string for ’GET’, ’SET’ etc.
DELAY=INTEGER	requested delay time in millisecond
EVENT=INTEGER	received message status
INVAL=CHARACTER*(MSG_VAL_LEN)	a received value string
LENGTH=INTEGER	number of bytes in VALUE
MAXVALS=INTEGER	maximum number of values
MESSID=INTEGER	transaction number
NAMECODE=INTEGER	parameter system code-number for the action
NVALS=INTEGER	actual number of values
OUTVAL=CHARACTER*(MSG_VAL_LEN)	a sent value string
PARKEY=CHARACTER*(PAR__SZNAM)	parameter keyword
PATH=INTEGER	path identifier to another task
REASON=INTEGER	symbol giving reason for reschedule
REQUEST=INTEGER	symbol requesting reschedules
RESULT=LOGICAL	.TRUE. implies an AST has occurred
SEQ=INTEGER	sequence number
STATUS=INTEGER	status
STRINGS(*)=CHARACTER*(*)	array of character strings
TASK_NAME=CHARACTER*(PAR__SZNAM)	name of another task
TIMEOUT=INTEGER	timeout in milliseconds, -1 = infinite
VALUE=CHARACTER*(MSG_VAL_LEN)	value string

```
*  Add to the list of active subsidiary actions for an action
CALL TASK_ADD_MESSINFO ( PATH, MESSID, STATUS )
Given : PATH, MESSID
Given and returned : STATUS

*  Used in application AST routine to signal to main-line code
CALL TASK_ASTMSG ( ACTNAME, LENGTH, VALUE, STATUS )
Given : ACTNAME, LENGTH, VALUE
Given and returned : STATUS

*  Used in application AST routine to signal to main-line code
CALL TASK_ASTSIGNAL ( ACTNAME, STATUS )
Given : ACTNAME
Given and returned : STATUS
```

```
*  Request a task to cancel an action
CALL TASK_CANCEL ( TASK_NAME, ACTKEY, INVAL, OUTVAL, STATUS )
Given : TASK_NAME, ACTKEY, INVAL
Given and returned : STATUS
Returned : OUTVAL

*  Concatenate an array of strings into an argument list
CALL TASK_CNCAT ( NVALS, STRINGS, VALUE, STATUS )
Given : NVALS, STRINGS
Given and returned : STATUS
Returned : VALUE

*  Wait for final acknowledgement from task
CALL TASK_DONE ( TIMEOUT, PATH, MESSID, OUTVAL, STATUS )
Given : TIMEOUT, PATH, MESSID
Given and returned : STATUS (Returns the status associated with the final
                     acknowledgement message from the task.)
Returned : OUTVAL

*  Get a parameter value from a task
CALL TASK_GET ( TASK_NAME, PARKEY, OUTVAL, STATUS )
Given : TASK_NAME, PARKEY
Given and returned : STATUS
Returned : OUTVAL

*  Get current action context
CALL TASK_GET_CONTEXT ( CONTEXT, STATUS)
Given and returned : STATUS
Returned : CONTEXT

*  Get current action context name
CALL TASK_GET_CONTEXTNAME ( CONTEXTNAME, STATUS)
Given and returned : STATUS
Returned : CONTEXTNAME

*  Get details of message which forced reschedule
CALL TASK_GET_MESSINFO ( PATH, CONTEXT, ACTKEY, VALUE, MESSID, EVENT, STATUS)
Given and returned STATUS
Returned : PATH, CONTEXT, ACTKEY, VALUE, MESSID, EVENT

*  Get current action name
CALL TASK_GET_NAME ( ACTNAME, STATUS)
Given and returned : STATUS
Returned : ACTNAME
```

```
*  Get parameter system code for current action name
CALL TASK_GET_NAMECODE ( NAMECODE, STATUS)
Given and returned : STATUS
Returned : NAMECODE

*  Get reason for current reschedule
CALL TASK_GET_REASON ( REASON, STATUS )
Given and returned : STATUS
Returned : REASON

*  Get current action sequence number
CALL TASK_GET_SEQ ( SEQ, STATUS)
Given and returned : STATUS
Returned : SEQ

*  Get value string for current action
CALL TASK_GET_VALUE ( VALUE, STATUS )
Given and returned : STATUS
Returned : VALUE

*  Signal another action to reschedule
CALL TASK_KICK ( ACTNAME, LENGTH, VALUE, STATUS )
Given : ACTNAME, LENGTH, VALUE
Given and returned : STATUS

*  Send an OBEY to a task
CALL TASK_OBEY ( TASK_NAME, ACTKEY, INVAL, OUTVAL, PATH, MESSID, STATUS )
Given : TASK_NAME, ACTKEY, INVAL
Given and returned : STATUS (Returns the status associated with the initial
                     acknowledgement message from the task.)
Returned : OUTVAL, PATH, MESSID

*  Set delay before next entry for current action
CALL TASK_PUT_DELAY ( DELAY, STATUS )
Given : DELAY
Given and returned : STATUS

*  Request the action to be rescheduled on certain events
CALL TASK_PUT_REQUEST ( REQUEST, STATUS )
Given : REQUEST
Given and returned : STATUS

*  Set current action sequence number
CALL TASK_PUT_SEQ ( SEQ, STATUS )
Given : SEQ
Given and returned : STATUS
```

```
*  Set value string for current action
CALL TASK_PUT_VALUE ( VALUE, STATUS )
Given : VALUE
Given and returned : STATUS

*  Set a parameter value in a task
CALL TASK_SET (TASK_NAME, PARKEY, INVAL, STATUS )
Given : TASK_NAME, PARKEY, INVAL
Given and returned : STATUS

*  Split an argument list into an array of strings
CALL TASK_SPLIT ( VALUE, MAXVALS, NVALS, STRINGS, STATUS )
Given : VALUE, MAXVALS
Given and returned : STATUS
Returned : NVALS, STRINGS

*  Return a triggering message to the controlling task
CALL TASK_TRIGGER ( ACTNAME, VALUE, STATUS )
Given : ACTNAME, VALUE
Given and returned : STATUS

*  Test interrupt flag
CALL TASK_TSTINTFLG ( RESULT, STATUS )
Given and returned : STATUS
Returned : RESULT
```

The following generic string-handling routines are provided to help in building or interpreting VALUE strings. Each TASK_xxx<T> routine represents the set of calls TASK_xxxC, TASK_xxxD, TASK_xxxI, TASK_xxxL, and TASK_xxxR.

NDIMS=INTEGER	number of dimensions
DIMS(*)=INTEGER	sizes of dimensions
NMAXDIMS=INTEGER	maximum number of dimensions
MAXDIMS(*)=INTEGER	maximum sizes of dimensions
STRING=CHARACTER*(*)	string being built or interpreted
<T>VAL=<TYPE>	value being converted
<T>VALS()=<TYPE>	array being converted
STATUS=INTEGER	status

```
*  Decode a character string as a value
CALL TASK_DEC0<T> ( STRING, <T>VAL, STATUS )
Given : STRING
Given and returned : STATUS
Returned : <T>VAL
```

```
*  Decode a character string as a vector
CALL TASK_DEC1<T> ( STRING, MAXVALS, NVALS, <T>VALS, STATUS )
Given : STRING, MAXVALS
Given and returned : STATUS
Returned : NVALS, <T>VALS

*  Decode a character string as an array
CALL TASK_DECN<T> ( STRING, NMAXDIMS, MAXDIMS, NDIMS, DIMS, <T>VALS, STATUS )
Given : STRING, NMAXDIMS, MAXDIMS
Given and returned : STATUS
Returned : NDIMS, DIMS, <T>VALS

*  Encode a value as a character string
CALL TASK_ENC0<T> ( <T>VAL, STRING, STATUS )
Given : <T>VAL
Given and returned : STATUS
Returned : STRING

*  Encode a vector as a character string
CALL TASK_ENC1<T> ( NVALS, <T>VALS, STRING, STATUS )
Given : NVALS, <T>VALS
Given and returned : STATUS
Returned : STRING

*  Encode an array as a character string
CALL TASK_ENCN<T> ( NDIMS, DIMS, <T>VALS, STRING, STATUS )
Given : NDIMS, DIMS, <T>VALS
Given and returned : STATUS
Returned : STRING
```

The following three generic routines are provided for compatibility with earlier ADAM releases. New applications should use the TASK_ENC calls instead.

```
CALL TASK_VAL0<T> ( <T>VAL, STRING, STATUS )
CALL TASK_VAL1<T> ( NVALS, <T>VALS, STRING, STATUS )
CALL TASK_VALN<T> ( NDIMS, DIMS, <T>VALS, STRING, STATUS )
```

E REQUEST constants

The values your application can set using TASK_PUT_REQUEST to instruct the fixed-part are

ACT__ASTINT	requests reschedule on AST receipt, with optional timer
ACT__CANCEL	signifies the action has completed due to a CANCEL
ACT__MESSAGE	requests reschedule on message receipt, optional timer
ACT__STAGE	requests an immediate (10msec) timed reschedule
ACT__WAIT	requests a timed reschedule

The following ACT__ constants also exist for upwards compatibility with ADAM Version 1.

ACT__END	signifies the action has completed successfully
ACT__EXIT	*** for U-task writers ONLY ***
ACT__INFORM	signifies the action has completed with an error
ACT__UNIMP	the requested action is not implemented

The fixed-part only acts on the REQUEST if your task returns a status of SAI__OK. Any other status returned, for example an error status of some sort, causes the action to be closed down. The error status is returned to the task which issued the original OBEY. Note that SS$_NORMAL is not recognised as an OK status, and will be reported as an error.

F Important STATUS values

The inquiry routines TASK_GET_REASON and TASK_GET_MESSINFO return a value in one of two groups. The first group concerns reschedules caused by events in this task.

MESSYS__RESCHED	a timer has expired
MESSYS__ASTINT	an AST routine has been obeyed
MESSYS__KICK	the action has been 'kicked'

The second group concerns messages received from a subsidiary task.

MESSYS__TRIGGER	a trigger message has arrived, the subsidiary action continues
DTASK__ACTCOMPLETE	the subsidiary action completed successfully
DTASK__ACTINFORM	the subsidiary action completed with an error
DTASK__ACTCANCEL	the subsidiary action was cancelled
DTASK__IVACTSTAT	the subsidiary action completed with an illegal REQUEST

In addition, the value can be any error status returned by the action in the subsidiary task.

G Document changes

There is a correction to the example CMOTASK in Appendix B and some re-wording of Appendix C.