Starlink Project
Starlink System Note 77.0

A.J. Chipperfield

16 August 2001

ADAM

The Control Subsystem

Abstract

This document describes the way an ADAM task interacts with the ADAM Message System (AMS). It is intended for system programmers.

The reader is expected to be familiar with the Starlink Software Environment documented in SG/4, SUN/144 and SUN/134

Contents

1 Introduction

ADAM tasks consist of user code wrapped up in the 'ADAM fixed part'. The top level of the user's code is a subroutine with the same name as the task and a single argument `STATUS` conforming to the normal Starlink inherited status strategy. The ADAM fixed part is provided by the task-building scripts, `alink` and `ilink`, and handles the interface with the outside world. The two main parts of the ADAM fixed part are:

The control subsystem provided by the DTASK and TASK libraries and the ADAM message system, AMS). This allows a task to be run directly from the shell, or to be controlled from a user-interface (or another task) and to control subsidiary tasks. It makes use of the TASK library which maintains information about the task in its COMMON blocks and provides user facilities for writing complete instrumentation systems with control tasks running subsidiary tasks. For more details on this, see ADAM – Guide to Writing Instrumentation Tasks (SUN/134).

The parameter subsystem provided by the SUBPAR, PARSECON and LEX libraries. This allows the task to obtain parameter values from a variety of sources, to output parameter values, to remember previous values and to share 'GLOBAL' parameter values with other tasks.

All the subroutine libraries involved are released as the Starlink Software Item PCS.

This document describes the control subsystem and the way it interacts with the message system and the parameter system.

2 The ADAM Main Routine

When a task is first started, `MAINTASK` (in `dtask_main.f`) finds out if it is being run via the ADAM message system or directly from the shell, by inspecting the environment variable `ICL_TASK_NAME`.

If `ICL_TASK_NAME` is not set, it is assumed that the task is being run direct from the shell – signal handling is disabled and `DTASK_DCLTASK()` is called. The parameter system is activated, the user's code is run through once and the parameter system de-activated, updating GLOBAL parameters if required. The program then exits. Any output messages or parameter prompts will go directly to `stdout` and replies to parameter prompts are read from `stdin` – the message system will take no part in the proceedings. This very simple running of a task is not discussed further in this document.

If `ICL_TASK_NAME` is set, it is assumed that the task is being run via the ADAM message system – signal handling is enabled (`DTASK_SETSIG()`) and `DTASK_DTASK()` is called to handle messages.

3 DTASK_APPLIC

`DTASK_APPLIC()` is called by any of the DTASK routines which need to activate the user's application code.

The code for `DTASK_APPLIC` is built into the ADAM task linking scripts, `alink` and `ilink`, and is modified by them to call the user's top-level subroutine.

`DTASK_APPLIC` copies information about the current action into the TASK common blocks, runs the user's code, runs down the parameter system if required and retrieves values, such as the action sequence number, the action value string, the delay before next action entry and the application request which may have been set by the user's code. These are passed back to the ADAM fixed part for appropriate action.

The code for `DTASK_APPLIC` differs between `alink` and `ilink` in that the process of closing down the parameter system, thus updating 'GLOBAL' parameters and resetting all parameters to the 'ground' state, does not occur with an `ilink`ed task. (It may also be prevented with an `alink`ed task by setting the environment variable `ADAM_TASK_TYPE` to 'I' in the task's process when the task is run.)

As can be seen from the above, although the user's code will probably differ markedly in structure between A-tasks, A-task monoliths, I-tasks and control tasks, as far as the ADAM fixed part is concerned there is very little difference between them. (There are further differences in the way the parameter subsystem handles things. These are triggered by the structure of the task's interface module.)

4 Receiving Control Messages

`DTASK_DTASK()` firstly calls `DTASK_PRCNAM()` to get the name by which the task is to register with the ADAM message system. This will be the name set in `ICL_TASK_NAME` by the controlling task – `DTASK_PRCNAM()` does not involve the message system. The task then registers with this name by calling `AMS_INIT()` (from `DTASK_INIT()`).

A loop is then entered, which continues until a bad `STATUS` is found. Within the loop most errors will be handled, but an MSP error probably means that no more messages can be sent or received so the task will exit.

Within the loop:

- The path for SUBPAR output is first set to 0 – *i.e.* the task is not handling a message from another task. There should be no output but if there is it will go to `stdout`.
- `AMS_RECEIVE()` with infinite timeout waits for a message – The message may come from a controlling task or user interface, or may be generated internally, by a timeout for example. Certain control messages such as connection requests are handled invisibly by `AMS_RECEIVE()` but it returns when it receives a normal message (message function `MESSYS__MESSAGE`), a new transaction is started and the message status (`MSGSTATUS`) indicates the type of message.

- The message is handled. This may involve sending and receiving further messages as part of the same transaction.
- The transaction is closed by sending a final acknowledgement with an appropriate message status.

5 Message Types

In a normal message (message function MESSYS__MESSAGE) received by an ADAM task, the message status MSGSTATUS indicates the action required.

MSGSTATUS = SAI__OK indicates the message is a 'GSOCC' message, the message context, CONTEXT indicates which one of:

GET Requesting a parameter value from the task.

SET Setting a parameter value for the task.

OBEY Requesting that an action is obeyed.

CANCEL Requesting that an action in a waiting state is cancelled.

CONTROL Requesting that a control function be performed.

Permitted control functions are:

DEFAULT to set the task's current working directory.

SETENV to set the value of an environment variable in the task's process.

PAR_RESET to reset all parameters, or parameters of a named action in a monolith, back to the 'ground' state. For a discussion of parameter states, see SUN/114.

N.B. DEFAULT and SETENV are required because tasks are run in separate processes, therefore (under Unix anyway) changing the current working directory or environment variables of the controlling process will not alter them for any existing subsidiary tasks.

See Handling GSOCC Messages (Section 6) for more details.

MSGSTATUS **not** SAI__OK may be:

MESSYS__EXTINT This should not happen in a task – the facility is for use by user interfaces. An error report is made and the event ignored.

MESSYS__RESCHED A timer has expired in this task and AMS_REMSG() has been called. DTASK_TIMEOUT() is called to handle the message. It checks that the specified action is active and that the reschedule was expected. If all is OK, the action is obeyed (DTASK_OBEY()).

MESSYS__ASTINT An AST has fired in this task and AMS_ASTINT() or AMS_ASTMSG() has been called. DTASK_ASTINT() checks the action and obeys it if OK (DTASK_OBEY()).

MESSYS__KICK Subroutine AMS_KICK() has been called by this task to run one of its actions. DTASK_KICK() checks the action and obeys it if OK (DTASK_OBEY()).

Other > 0 May be from a subsidiary task. See Messages from Subsidiary Tasks (Section 7) for more details.

MSGSTATUS < 0 An MSP error, set STATUS to terminate loop.

6 Handling GSOCC Messages

These are handled by DTASK_GSOC() in the first instance. In addition to the transaction identification, PATH and MESSID, the CONTEXT, NAME and VALUE components of the received message are passed.

Firstly the PATH and MESSID from the message are set for SUBPAR (SUBPAR_PUTPATH()) in case it wants to reply with a message or request a parameter value as a result of this message, then a DTASK routine appropriate to the CONTEXT is called.

When the required action for the CONTEXT is complete, the transaction is ended by calling (DTASK_COMSHUT()) which flushes any error messages on the stack to the master task (using ERR_CLEAR()) before sending a final acknowledgment. (The master task must therefore be prepared to handle any MESSYS__INFORM messages prior to the final acknowledgement in any CONTEXT.) The final acknowledgement will have an appropriate message status and the message value will depend upon the status and the CONTEXT.

6.1 GET Context

GET is handled by DTASK_GET(). NAME contains the name of the required parameter. If the task is a monolith, the parameter name must be of the form ACTION:PARNAME to specify the particular action within the monolith.

The value of the named parameter is obtained, by searching the vpath if the parameter is not already active.

Note that there are some restrictions on the vpath search:

- Dynamic will usually have no effect because any dynamic defaults will be cancelled initially and if the parameter was reset.
- Prompting is disabled so any attempt to prompt will return status PAR__NOUSR.

The transaction is then closed (DTASK_COMSHUT()). Assuming the status is SAI__OK, the message value of the final acknowledgement is the parameter value as a character string. In the event of an error, error messages are reported and the message status set appropriately.

6.2 SET Context

SET is handled by DTASK_SET(). NAME contains the name of the required parameter as for GET. VALUE is a character string which the parameter subsystem can resolve into a value for the specified parameter. The given value is put into the parameter's internal storage and the parameter is made active (SUBPAR_CMDPAR()). The transaction is then closed (DTASK_COMSHUT()). If an error occurred, error messages will be reported and an appropriate status set in the final acknowledgement.

6.3 OBEY and CANCEL Context

OBEY and CANCEL are further checked in DTASK_GSOC().

NAME contains the name of the required action (or task within a monolith) and VALUE is a string specifying parameter values in a format acceptable to the parameter system.

For OBEY the action must not be active and for CANCEL the action must be active; otherwise the transaction is terminated (in DTASK_COMSHUT()) by calling AMS_REPLY() with a suitable bad message status.

If the OBEY/CANCEL is valid, SUBPAR_FINDACT() is called to set up the parameter system for the specified action and the VALUE string is processed (SUBPAR_CMDLINE()) to set the value of any given parameters.

6.3.1 OBEY

The new action is added to the active list and the list of subsidiary tasks for it is cleared. AMS_REPLY() is called with message status DTASK__ACTSTART to send an initial acknowledgement of the OBEY message .

If that fails the action and/or transaction are closed (DTASK_ACTSHUT() or DTASK_COMSHUT()); if it is OK, DTASK_OBEY() is called.

DTASK_OBEY() calls DTASK_APPLIC() with context OBEY and that in turn calls the user's top-level routine.

On return from DTASK_APPLIC() if STATUS is not SAI__OK, the message status, MESSTATUS, for the final acknowledgement message is set to the bad status value; if STATUS is SAI__OK, the REQUEST returned from the application is checked (DTASK_ACT_SCHED()) to see whether the action is to be terminated or is to be rescheduled. If the user's code has not changed it, the REQUEST will be ACT__END.

If a reschedule is requested, DTASK_ACT_SCHED() will set it up; otherwise the action is ended and the communications transaction closed by sending a reply to the task which initiated the OBEY. MESSTATUS for the reply depends on the STATUS and REQUEST returned from DTASK_APPLIC().

If a reschedule was not requested, MESSTATUS will depend upon the value of REQUEST as follows:

REQUEST	MESSTATUS
ACT__END	DTASK__ACTCOMPLETE
ACT__UNIMP	DTASK__UNIMP
ACT__INFORM	DTASK__ACTINFORM
SAI__OK	DTASK__IVACTSTAT
DTASK__SYSNORM	DTASK__IVACTSTAT
ACT__CANCEL	DTASK__IVACTSTAT
Other	REQUEST

Error messages may also be reported.

6.3.2 CANCEL

CANCEL is handled by calling DTASK_CANCEL().

Details of the OBEY transaction which started this action are obtained from the COMMON blocks (DTASK_CMN) then DTASK_APPLIC() is called with context CANCEL and that in turn calls the user's code. See SUN/134 for examples of how the user's code might handle a CANCEL

On return from DTASK_APPLIC the status and REQUEST returned from the application are checked (DTASK_ACT_SCHED()) to see whether the action is to be terminated or is to continue rescheduling. In any case, an acknowledgement is sent to the task which requested the CANCEL. If the action has ended, an acknowledgement is also sent to the task which issued the OBEY.

6.4 CONTROL Context

CONTROL is handled by DTASK_CONTROL(). The message name component, passed as ACTION defines the particular control function required, 'DEFAULT', 'SETENV' or 'PAR_RESET'. The required action is performed.

If a problem occurs in the process, an appropriate bad status value is set and error report is made.

The transaction is then closed (DTASK_COMSHUT()).

6.4.1 DEFAULT

The message value specifies a new 'current working directory' required for the task. If the value is blank, no change is required; otherwise the change is made by calling the appropriate system routine.

Whether or not a change was made, the 'new' current working directory is obtained and returned as the message value in final reply message. CONTROL DEFAULT can therefore be used to enquire the task's current working directory without changing it.

6.4.2 SETENV

The message value specifies a new value for an environment variable for the task. It has the form: 'VARIABLE = NEWVALUE', where VARIABLE is the name of the environment variable and NEWVALUE is the required value. The string may be quoted – quotes will be stripped in the usual way. If the string is unquoted, leading blanks will also be stripped from NEWVALUE.

The environment variable is set by calling the appropriate system routine.

6.4.3 PAR_RESET

This causes the task's parameters to be reset to the 'ground' state. The message value may be used to specify the name of an individual task in a monolith; if it doesn't, parameters for all tasks in the monolith are reset. Resetting is done by calling SUBPAR_DEACT() with argument TTYPE set to 'R' – MIN, MAX and dynamic default values are also cancelled but GLOBAL values are not updated.

7 Messages from Subsidiary Tasks

DTASK_SUBSID() is called to check it. Given the PATH and MESSID of the message, the Active Subsidiary Task Action Block (see the TASK library) is searched and the associated action in this task found (or an error). Communications to the parent task for the associated action are set for SUBPAR (SUBPAR_PUTPATH()), then the particular message is handled depending upon the MSGSTATUS.

MESSYS__INFORM is relayed on to this task's parent (using SUBPAR_WRITE()).

MESSYS__PARAMREQ is handled by TASK_ASKPARAM(). It relays the parameter request on to this task's parent (using SUBPAR_REQUEST() which waits for the MESSYS__PARAMREP reply and returns the value). TASK_ASKPARAM() then returns the value in a MESSYS__PARAMEREP to the subsidiary task.

MESSYS__SYNC causes this task to synchronise (see Section 9) (in SUBPAR_SYNC()) with its parent and then acknowledge the original message with a MESSYS__SYNCREP to the subsidiary task (using AMS_REPLY()). See

The above are said to be 'transparent' messages. If the message is not one of these, the message information is put (TASK_PUT_MESSINFO()) into the Current D-task Action Block in case the associated action requires it later.

If there is a failure in any of the above attempts to relay messages, the subsidiary task is removed from the Active Subsidiary Task Action Block and the action in this task obeyed (DTASK_OBEY()) again.

MESSYS__TRIGGER causes the associated action in this task to be obeyed (DTASK_OBEY()) as if it were initiated by its parent.

Other MSGSTATUS > 0 is assumed to mean that action is complete in the subsidiary task – the subsidiary task action is cleared from the Active Subsidiary Task Action Block and, again DTASK_OBEY() is called. Failure to relay and other terminations are therefore treated the same.

8 Re-scheduling

An action being obeyed or cancelled can request that it is re-entered (re-scheduled) upon some future event by calling TASK_PUT_REQUEST() and optionally TASK_PUT_DELAY(). See SUN/134 for details.

Any values of the request and delay which have been set are returned to the calling routine (DTASK_OBEY or DTASK_CANCEL by DTASK_APPLIC and DTASK_ACT_SCHED is called to check if a reschedule is required and set it up if required.

Reschedule requests may be:

ACT__STAGE Re-schedule immediately (in fact after 10ms).

`ACT__WAIT` Re-schedule after `DELAY` ms.

`ACT__ASTINT` Re-schedule on AST (with timeout of `DELAY`ms).

`ACT__MESSAGE` Re-schedule on completion of an action in a subsidiary task (with timeout of `DELAY`ms).

For `ACT__ASTINT` and `ACT__MESSAGE`, `DELAY` may be `MESSYS__INFINITE` indicating that no timer should be set.

As each reschedule is set, the action sequence number, initially 0, is incremented.

Where a timeout is required, an ATIMER timer is set by:

```
CALL DTASK_RESCHED( ACTPTR, ACTCNT, SCHEDTIME, STATUS )
```

`ACTPTR` specifies the action to be rescheduled, and `ACTCT` the timer identifier. The timer is set to call `DTASK_CHDLR()` after `SCHEDTIME` millisecs. `DTASK_CHDLR()` is written in C and is a straight-through call to `DTASK_ASTHDLR()` which calls `AMS_RESMSG()` to send a reschedule message (message status `MESSYS__RESCHED`) to this task.

When `DTASK_DTASK()` receives the reschedule message, it calls `DTASK_TIMEOUT()` which deconstructs the passed value into an action pointer and count, and checks that the specified action reschedule was expected.

If that checks OK, the action is obeyed (`DTASK_OBEY()`); if not, it is ignored.

9 Synchronisation

If the task needs to output directly to the user's terminal, to switch it to 'graphics' mode for example, it is necessary to ensure that the user-interface has completed output of previously sent text messages before the graphics output is sent. This is achieved by the user's code calling `MSG_SYNC()` which sends a `MESSYS_SYNC` message to the master task and waits for a `MESSYS_SYNCREP` message to be returned. As all messages are queued by the ADAM message system, we can be sure that all earlier message have been handled by the time the task receives the reply. *Note that this only works where no other tasks can send messages to the user-interface in the meantime.*

Where a control task lies between a subsidiary task and the user-interface, the messages are simply relayed.

10 Use of the TASK Library

The TASK library maintains two lists in COMMON: the Current D-task Action Block and the Active Subsidiary Task Action Block. It also maintains the AST interrupt flag, indicating if `TASK_ASTSIGNAL()` has been called.

The Current D-task Action Block holds details of the current D-task action, *i.e.* the one on whose behalf the current call to ACT has been made. Items held are:

- Action pointer for current action
- Message path resulting in current entry
- Message id resulting in current entry
- Context in message resulting in entry
- Status in message resulting in entry
- Context for current action
- Name code for current action
- Sequence number for current action
- Delay before next entry
- Request for rescheduling
- Current action name
- Action name in message
- Value string for current action

The Active Subsidiary Task Action Block holds details of active actions that have been initiated in subsidiary tasks. If an incoming message corresponds to one of the actions described in this common block, it will result in an entry to ACT. Items held are (for each subsidiary action):

- Action pointer for initiating action (in this task)
- Message path for initiated action
- Message id for initiated action

A Typical A-task Message Sequence

User Interface		Task
Register with message system		
Load task		
		Register with message system
		Await message
Send connection request	>	
	<	Acknowledge connection request (automatically)
		Wait for next message
Send GSOCC OBEY	>	
	<	Acknowledge GSOC OBEY
		Obey user's code

The user's code may, potentially, generate any number of:

	<	Reply with `MESSYS__INFORM` (output from `MSG_OUT()`, `ERR_FLUSH()` *etc.*)
Display message value to user		

or:

	<	Reply with `MESSYS__PARAMREQ` (Request a prompt for a parameter value)
Prompt User		
Reply with `MESSYS__PARAMREP`	>	
		Set parameter value

or:

	<	Reply with `MESSYS__SYNC` (Output from `MSG_SYNC`)
Prompt user		
Reply with `MESSYS__SYNCREP`	>	

and finally, from the ADAM fixed part:

Terminate transaction (automatically) Display final status if bad.	<	Reply with terminating message status (`DTASK__ACTCOMPLETE` is good)