SUN/241.0

Starlink Project
Starlink User Note 241.0

B D Kelly (ROE)
A J Chipperfield (RAL)

16 August 2001

AMS
The Unix ADAM Message System
2.0
Programmer's Manual

Abstract

The ADAM Message System (AMS) library, which implements the ADAM inter-task communications protocol under Unix, is described, along with its Fortran-callable interface (FAMS).

The description of AMS is distinguished from the current implementation which uses the Message System Primitives (MSP).

Contents

1 Introduction **1**

2 Transactions **1**

3 Task Initialisation **1**

4 Task Exit **2**

5 Opening Communications **2**

6 Sending a Command **2**

7 Getting Expected Replies **3**

8 Receiving a Command **3**

9 Sending Expected Replies **4**

10 Sending Internal Messages **4**

11 Implementation **4**
11.1 AMS Messages 4
11.2 MSP, Sockets and Queues 5
11.3 Communications Directory 6
11.4 MSP Messages 6

A Example **7**

B Function Descriptions **12**
AMS_ASTINT 13
AMS_ASTMSG 14
AMS_EXTINT 15
AMS_GETREPLY 16
AMS_INIT 17
AMS_INITEH 18
AMS_KICK 19
AMS_PATH 20
AMS_PLOOKUP 21
AMS_RECEIVE 22
AMS_REPLY 23
AMS_RESMSG 24
AMS_SEND 25

1 Introduction

The AMS library enables communications to be opened between ADAM tasks, and commands and acknowledgements to be sent and received. Provided suitable ADAM network (ADAMNET) processes are loaded, communications can also take place across networks. In the absence of ADAMNET processes, communication is restricted to ADAM tasks on a single machine.

The example (see Section A) shows that communicating programs do not have to be strictly ADAM tasks but the term 'task' is used throughout this document to mean either end of a communications link. More details on the way ADAM uses AMS can be found in SSN/77

AMS is currently implemented using the Message System Primitives (MSP) and the ADAM Timer (ATIMER) libraries. AMS, MSP and ATIMER are all written in C and included in the Parameter and Communication Subsystems PCS Starlink Software Item. Fortran interfaces (FAMS and FATIMER) are also provided for AMS and ATIMER respectively.

This document describes AMS and its current implementation, keeping the two separate as far as possible in order to clarify the distinction whilst giving readers a feel for the way the whole system currently works.

2 Transactions

AMS communications are carried out as a series of 'transactions'. A transaction consists of an initial message and a number of further messages (replies) in either direction, associated with the initial message. Separate transactions are used to set up a communications path and to carry out the business of obeying a command. A command transaction is started by calling `ams_send` and is terminated as described under 'Getting Expected Replies' (see Section 7).

When sending a message, the user specifies a `message_function` and a `message_status` which are passed as arguments to functions `ams_send` or `ams_reply`.

The value of `message_function` may be:

MESSYS__INIT Used to ask for a communications link to another task. (`ams_send` only)[1].

MESSYS__DE_INIT Used to close a communications link to another task.

MESSYS__MESSAGE Used for all other purposes and qualified by the `message_status` and possibly other arguments.

3 Task Initialisation

A task initialises AMS by calling `ams_init()` specifying the name by which the task is to be known to the message system. The name is registered and an exit handler set up. Sometimes it is not desirable to set up an exit handler – in that case `ams_initeh()` should be used.

[1]This function of `ams_send` is rendered obsolete by `ams_path`.

A controlling task will need to know the name by which the subsidiary task is known to the message system. (For ADAM tasks this is done by having the user interface set environment variable `ICL_TASK_NAME` to the required name. This also serves to tell the task that it is indeed being run via the ADAM message system and not directly from the Unix shell.)

4 Task Exit

When a task receives a signal causing it to exit, the message system exit handler is invoked (assuming it has been set up). The result is that messages are sent to any other tasks which have been in communication with the exiting task, informing them of its removal, and the task name is de-registered from the message system.

5 Opening Communications

Once a task has initialised itself successfully, it can open communications with another initialised task by using `ams_path()`, specifying the name by which the other task is registered. The path number returned can then be used for further communications with the other task.

A short transaction consisting of a connection request message and an acknowledgement from the slave task is carried out. If the slave task is waiting to receive a command (see Section 8), the acknowledgement is sent automatically from within `ams_receive()`. The acknowledgement is expected within `MESSYS__INIT_WAIT_TIME` (currently 30000) milliseconds.

6 Sending a Command

Having obtained a path, `ams_send()` can be used to send a command message to the task identified by that path – the `message_function` argument is set to `MESSYS__MESSAGE`. If `ams_send()` succeeds, it starts a new transaction and returns a transaction id (`messid`) which remains valid for the duration of the transaction. The task which sends the message becomes the 'master' and the one which receives it becomes the 'slave' for the transaction.

Command messages are designed for use with ADAM tasks but this does not preclude their use for other purposes (see the example in Section A). The other arguments to `ams_send` are packed into the message and are unpacked by AMS at the other end. What the slave task does with them is up to it – AMS generally has no interest in the other arguments of `ams_send()`.

There is one exception to this – if the `message_context` is `OBEY`, the receiving end will allow the transaction to include further messages from the master task as part of the same transaction. This enables the slave to reply with requests (prompts) for parameter values and receive replies from the master.

ADAM use of the other arguments is as follows:

`message_status` Should be SAI__OK.

`message_context` One of the following constants, defined in `adamdefns.h`:

`GET` To request a parameter value.

`SET` To set a parameter value.

`OBEY` To obey an action in the task.

`CANCEL` To cancel an action in the task.

`CONTROL` To control/enquire the task environment.

`message_name` Name of task action or parameter.

`message_length` Length of `message_value`.

`message_value` Context-dependent values.

7 Getting Expected Replies

Once a command is in progress between two tasks, they can receive replies specific to that command (as identified by `path,messid`) by using `ams_getreply()`.

`ams_getreply()` ignores all messages not associated with the specified (`path,messid`) except those generated by the `ams_extint()` function.

For ADAM tasks, expected values of `message_status` are:

MESSYS__PARAMREQ Request a parameter value.

MESSYS__PARAMREP Reply to a PARAMREQ.

MESSYS__INFORM Message to be displayed

MESSYS__SYNC Synchronisation request

MESSYS__SYNCREP Synchronisation reply

MESSYS__TRIGGER Trigger an action in the master task

DTASK__ACTSTART [2] Acknowledge a GSOC command.

Completion of the command transaction and freeing of the associated `messid` occurs automatically if the task returns a `message_status` which is not one of the above. Apart from that, the other arguments returned by `ams_getreply` are of no significance to AMS.

8 Receiving a Command

Any message, including commands or replies associated with an existing command, can be received using `ams_receive()`. When a new command message is received a new transaction (`messid`) will be set up in the receiving task. The `path` and `messid` associated with the command/reply are returned.

See 'Sending a Command' (Section 6) for a discussion of the significance of the other arguments of `ams_receive`.

[2] This reference to DTASK, the main part of the ADAM task fixed part, is for historical reasons.

9 Sending Expected Replies

A reply associated with an existing command has to be sent using `ams_reply()`, specifying the relevant (`path,messid`). The command transaction is terminated automatically if a reply is sent with a `message_status` other than those listed in 'Getting Expected Replies' (Section 7)

See also 'Sending a Command' (Section 6) for a discussion of the significance of the other arguments of `ams_reply`.

10 Sending Internal Messages

A set of routines are provided to enable a task to send messages to itself. `ams_kick()` is intended to enable main-line code to queue a message which it can subsequently collect by using `ams_receive()`.

The other functions are intended only for calling from within signal handlers. `ams_resmsg()`, `ams_astmsg()` and `ams_astint()` generate messages to be read by `ams_receive()` but ignored by `ams_getreply()`.

`ams_extint()` generates a message which can be received by either `ams_receive()` or `ams_-getreply()` (its main purpose is to allow user interfaces to be implemented).

At the receiving end, these messages result in the appropriate value of `message_status`, one of:

MESSYS__KICK

MESSYS__RESCHED

MESSYS__ASTINT

MESSYS__EXTINT

11 Implementation

11.1 AMS Messages

AMS messages may be transported within other structures such as MSP messages but the AMS message itself consists of a message type, followed by a structure dependent upon the type. The types are defined in `ams_sys.h` and the associated structures in `ams_struc.h`. The message type name has three elements separated by '_':

LOC/REM Whether local or remote (this or other machine).

type **ACK** Acknowledge

GSOC_START Start GET/SET/OBEY/CANCEL/CONTROL

GSOC_END End GET/SET/OBEY/CANCEL/CONTROL

MSG Message.

DEINIT De-initialise

INIT Initialise

CALL Request remote connection (REM only)

ACCEPT Accept remote connection (REM only)

IN/OUT Whether message is in or out

E.g. LOC_ACK_IN, REM_CALL_OUT.

Internal messages are of type LOC_MSG_IN/OUT

The AMS functions will automatically send the right type of message, in the case of `ams_send()` and `ams_reply()` this will depend upon their `message_function` and `message_status` arguments.

11.2 MSP, Sockets and Queues

MSP uses a single STREAM socket (path) in the INET domain to communicate between any two tasks and a STREAM socket pair in the UNIX domain to send/receive local (internal) messages. Messages on the sockets specify a 'queue' in the receiving task for which they are intended and a queue in the sending task to which replies should be sent.

The MSP message reading function accepts a list of 'receive' queues on which it should look for messages and the MSP message sending function accepts a 'send' queue which specifies a socket and a receive queue in the other task. N.B. the reply queue may be given as `MSP__NULL_SENDQ` (it is a send queue in the task which has to reply).

AMS uses separate MSP queues within each task for:

- Receiving unexpected messages from other tasks (the command queue).
- Sending unsolicited messages to other tasks (other task's command queue).
- Receiving replies to a sent message which initiates a transaction – each transaction has its own queue (reply queue).
- Sending replies as part of a transaction (other task's reply queue)
- Sending local (internal) messages (one for each internal message type).

  sigast_q

  sigext_q

  sigkick_q

  sigresch_q

  sigtimeout_q

- Receiving local (internal) messages (one for each internal message type).

astint_q
extint_q
kick_q
resched_q
timeout_q

The AMS user will not need to know about MSP queues.

11.3 Communications Directory

MSP communications are opened between tasks by name. The name lookup is provided by each task making an entry in a directory pointed to by the environment variable `ADAM_USER`. If `ADAM_USER` is not defined when a task attempts to register itself with the message system, directory `~/adam` will be used and, if the specified directory does not exist, an attempt will be made to create it.

This results in a file being created in the `ADAM_USER` directory with a name compounded of the task name and an identifying number (*e.g* `$ADAM_USER/slave_5001`). Another task can then open communications by searching `ADAM_USER` for the right name and using the identifying number.

When a task exits, the AMS exit handler de-registers the task from MSP and the task's file is removed from `ADAM_USER`. If the task does not exit normally for some reason, the file in `ADAM_USER` may get left behind. In this case the file must be deleted explicitly; otherwise the task will refuse to load next time.

11.4 MSP Messages

MSP messages are passed between processes and also added to a queue at the receiving end. Some of the structure components are only relevant when it is being passed, or when it is on a queue.

A received MSP message contains the following information:

- Target queue in this task
- Reply queue in other task
- Actual size of message body
- Message body
- Pointer to next message in the queue

The content of the message body is only understood at higher levels (AMS in this case) – it is immaterial to MSP.

A Example

The example consists of a pair of C programs called `master` and `slave`. They should be run in the background by:

```
% slave &
% master &
```

The code for `master` is:

```
/* amsmaster
 * A test of ams - run in conjunction with amsslave
 *   % amsslave &
 *   % amsmaster
*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include <string.h>
#include <signal.h>
#include <errno.h>
#include <unistd.h>

#include "sae_par.h"
#include "adam_defns.h"
#include "messys_len.h"
#include "messys_par.h"

#include "ams.h"

int main()
{
   int outmsg_status;
   int outmsg_function;
   int outmsg_context;
   int outmsg_length;
   char outmsg_name[32];
   char outmsg_value[MSG_VAL_LEN];
   int inmsg_status;
   int inmsg_context;
   int inmsg_length;
   char inmsg_name[32];
   char inmsg_value[MSG_VAL_LEN];
   int status;
   int path;
   int messid;
   int j;

   status = 0;

/* Set up components of a GSOC OBEY message. The slave does not care about
```

```
 * the name component of the message */
   outmsg_status = SAI__OK;
   outmsg_function = MESSYS__MESSAGE;
   outmsg_context = OBEY;
   outmsg_length = 16;

   strcpy ( outmsg_name, "junk" );
   strcpy ( outmsg_value, "master calling" );

/* Register as "master" with the message system */
   ams_init ( "master", &status );
   if ( status != SAI__OK )
   {
      printf ( "master - bad status after ams_init\n" );
   }

/* Get a path to "slave" and report */
   ams_path ( "slave", &path, &status );
   if ( status != SAI__OK )
   {
      printf ( "master - bad status after ams_path\n" );
   }
   else
   {
      printf ( "master - path set up ok\n" );
   }

/* Perform 1000 identical transactions - send a GSOC obey message and
 * await an initial acknowledgement (message_status = DTASK__ACTSTART)
 * and a completion message (message_status = SAI__OK) */
   for ( j=0; j<1000; j++ )
   {
/* Send the OBEY command */
      ams_send ( path, outmsg_function, outmsg_status, outmsg_context,
        outmsg_name, outmsg_length, outmsg_value, &messid, &status );
/* Get the acknowledement reply - content not checked */
      ams_getreply ( MESSYS__INFINITE, path, messid, 32, MSG_VAL_LEN,
        &inmsg_status, &inmsg_context, inmsg_name, &inmsg_length,
        inmsg_value, &status );
/* Get the completion reply - content not checked.
 * AMS will terminate the transaction if it is the expected message status
 * SAI__OK */
      ams_getreply ( MESSYS__INFINITE, path, messid, 32, MSG_VAL_LEN,
        &inmsg_status, &inmsg_context, inmsg_name, &inmsg_length,
        inmsg_value, &status );
   }

/* If all OK, display the last received message value;
 * otherwise display the error status */
   if ( status != 0 )
   {
      printf ( "master: bad status = %d\n", status );
   }
   else
```

```
   {
      printf ( "master: received - %s\n", inmsg_value );
   }

   return 0;
}
```

The code for `slave` is:

```
/* amsslave
 * A test of ams - run in conjunction with amsmaster
 *   % amsslave &
 *   % amsmaster
*/
#include <stdio.h>
#include <time.h>
#include <sys/types.h>
#include <sys/time.h>
#include <string.h>
#include <signal.h>
#include <errno.h>
#include <unistd.h>

#include "sae_par.h"
#include "adam_defns.h"
#include "dtask_err.h"             /* dtask error codes */

#include "messys_len.h"
#include "messys_par.h"

#include "ams.h"

int main()
{
   int outmsg_status;
   int outmsg_function;
   int outmsg_context;
   int outmsg_length;
   char outmsg_name[32];
   char outmsg_value[MSG_VAL_LEN];
   int inmsg_status;
   int inmsg_context;
   int inmsg_length;
   char inmsg_name[32];
   char inmsg_value[MSG_VAL_LEN];

   int status;
   int path;
   int messid;
   int j;

/* Set up components of a reply to a GSOC OBEY message. The master does not
 * care about the name component of the message */
   status = 0;
```

```
   outmsg_status = SAI__OK;
   outmsg_function = MESSYS__MESSAGE;
   outmsg_context = OBEY;
   outmsg_length = 16;

   strcpy ( outmsg_name, "junk" );
   strcpy ( outmsg_value, "slave replying" );

/* Register as "slave" with the message system */
   ams_init ( "slave", &status );
   if ( status != 0 )
   {
      printf ( "slave: failed init\n" );
   }

/* Receive 1000 commands, sending an initial acknowledgement and a completion
 * message in reply to each */
   for ( j=0; j<1000; j++ )
   {
/* Await a command message */
      ams_receive ( MESSYS__INFINITE, 32, MSG_VAL_LEN, &inmsg_status,
        &inmsg_context, inmsg_name, &inmsg_length, inmsg_value, &path,
        &messid, &status );

/* Send an initial acknowledgement (message_status = DTASK_ACTSTART). */
      outmsg_status = DTASK__ACTSTART;
      ams_reply ( path, messid, outmsg_function, outmsg_status,
        outmsg_context, outmsg_name, outmsg_length, outmsg_value,
        &status );

/* Send a completion message (message_status = SAI__OK) - this will terminate
   the transaction at both ends */
      outmsg_status = SAI__OK;
      ams_reply ( path, messid, outmsg_function, outmsg_status,
        outmsg_context, outmsg_name, outmsg_length, outmsg_value,
        &status );

/* If there was a failure, exit the loop */
      if ( status != SAI__OK ) break;

   }

/* If all OK, display the last received message value;
 * otherwise display the error status */
   if ( status != 0 )
   {
      printf ( "slave: bad status = %d\n", status );
   }
   else
   {
      printf ( "slave: received - %s\n", inmsg_value );
   }

   return 0;
```

```
}
```

B Function Descriptions

AMS_ASTINT
Send an ASTINT message to this task

Invocation:

```
(void) ams_astint( *status )
```

Arguments:

status = int * (given and returned)
global status

Description:
Causes the task to send an ASTINT message to itself. This should be called from a signal handler

Implementation:
Send an OBEY message to the astint queue.

AMS_ASTMSG
Send an ASTMSG to this task

Invocation:

```
(void) ams_astmsg( name, length, value, status )
```

Arguments:

name = char * (given)
name of the action to be rescheduled

length = int (given)
number of significant bytes in value

value = char * (given)
message to be passed to main-line code

status = int * (given and returned)
global status

Description:
Causes the task to send specified ASTINT message 'value', qualified by 'name' to itself. This should be called from a signal handler.

Implementation:
Send an OBEY message to the astint queue.

AMS_EXTINT
Send an EXTINT message to this task

Invocation:

```
(void)ams_extint( status )
```

Arguments:

status = int * (given and returned)
global status

Description:

Causes the task to send an EXTINT message to itself. This should be called from a signal handler when an external interrupt has occurred.

N.B. This is not be used within a normal ADAM task - it is intended for use in user interfaces.

Implementation:

Send an OBEY message to the extint queue.

AMS_GETREPLY
Receive a message on a specified path, messid

Invocation:

```
(void)ams_getreply( timeout, path, messid, message_name_s,
message_value_s, message_status, message_context, message_name,
message_length, message_value, status )
```

Arguments:

timeout = int (given)
timeout time in milliseconds

path = int (given)
pointer to the path

messid = int (given)
message number of incoming message

message_name_s = int (given)
maximum space for name (bytes)

message_value_s = int (given)
maximum space for value (bytes)

message_status = int * (returned)
message status

message_context = int * (returned)
message context

message_name = char * (returned)
message name

message_length = int * (returned)
length of value

message_value = char * (returned)
message value

status = int * (given and returned)
global status

Description:

The application has sent a message on path 'path' as part of transaction 'messid' and wishes to obtain the reply within 'timeout' milliseconds. A timeout value of MESSYS__INFINITE indicates no time limit.

Any received message is unpacked appropriately and the contents returned to the calling routine. Only those arguments relevant to the particular message type will be returned.

Note that the received message may be a TIMEOUT message as a result of the timer being set.

Implementation:

The function first checks the transaction (messid) is legally identified and that there exists an acknowledge queue for that transaction.

If 'timeout' is not MESSYS__INFINITE, it then sets the timer clock going so that we get a timeout if there is no response within timeout milliseconds. The ATIMER package is used to handle timers.

The function then looks for a message on either the external interrupt queue, the transaction acknowledge queue or the timeout queue.

AMS_INIT
Initialise ams and register an exit handler

Invocation:

```
(void)ams_init( own_name, status )
```

Arguments:

own_name = char * (given)
name of this task

status = int * (given and returned)
global status

Description:
Initialise ams and register an exit handler

Implementation:
Call ams_initeh, requesting that an exit handler be set up.

AMS_INITEH
Initialise ams and optionally register an exit handler

Invocation:

```
(void)ams_initeh( own_name, eh, status )
```

Arguments:

own_name = char * (given)
name of this task

eh = int (given)
whether to register exit handler

status = int * (given and returned)
global status

Description:
Initialise AMS optionally registering an exit handler

Implementation:

Initialise the internal data structures.

Register with msp, obtain the command queue for incoming messages, then create the queues used for this task sending messages to itself.

Finally set up the signal handler if so requested.

AMS_KICK
Send a KICK message to this task.

Invocation:

```
(void)ams_kick( name, length, value, status )
```

Arguments:

name = char * (given)
name of the action to be rescheduled

length = int (given)
number of significant bytes in value

value = char * (given)
message to be passed to application code

status = int * (given and returned)
global status

Description:
Sends a KICK message to this task, specifying an action to be obeyed and a message (command line) to be passed to the application code.

Implementation:
Send a soft kick interrupt message 'value' qualified by 'name' to kick queue.

AMS_PATH
Get a communications path to another task

Invocation:

```
(void)ams_path( other_task_name, path, status )
```

Arguments:

other_task_name = char * (given)
name of task to which path is required

path = int * (returned)
the path number

status = int * (given and returned)
global status

Description:

Open a path to the task whose name is 'other_task_name' and return the path index in 'path'. The other task may be local or remote, indicated by a name of the form machine::name, where :: may be any of the permitted separator pairs and defines the ADAMNET process to be used.

Implementation:

A temporary transaction acknowledge queue is obtained, a MESSYS__INIT message sent via the path just obtained, and the reply obtained.

If this short transaction fails to complete, the path and any associated transactions are freed; otherwise the path (index) is returned.

AMS_PLOOKUP
Look up a taskname given a path to it

Invocation:

```
(void)ams_plookup( path, name, status )
```

Arguments:

path = int (given)
the path number

name = char * (returned)
the task name

status = int * (given and returned)
global status

Description:

Returns the name of the task connected via the given path.

If the task is remote, a name of the form xxxxx::yyyyyy is returned, where xxxxxx:: is the machine and ADAMNET indicator, and yyyyyy is the task name.

Implementation:

Given a path 'path', we check that the path is legal and then use it to ascertain whether the path is linked to a remote task or a local task. A name of the appropriate form is returned.

AMS_RECEIVE
Receive any incoming message

Invocation:

```
(void)ams_receive( timeout, message_name_s, message_value_s,
message_status, message_context, message_name, message_length,
message_value, path, messid, status )
```

Arguments:

timeout = int (given)
timeout time in milliseconds

message_name_s = int (given)
maximum space for name (bytes)

message_value_s = int (given)
maximum space for value (bytes)

message_status = int * (returned)
message status

message_context = int * (returned)
message_context

message_name = char * (returned)
message name

message_length = int * (returned)
length of value

message_value = char * (returned)
message value

path = int * (returned)
path on which message received

messid = int * (returned)
message number of incoming message

status = int * (given and returned)
global status

Description:

Looks for a message to this task from any source for 'timeout' milliseconds. A timeout value of MESSYS__INFINITE indicates no time limit.

Any received message is unpacked appropriately and the contents returned to the calling routine. Only those arguments relevant to the particular message type will be returned.

Note that the received message may be a TIMEOUT message as a result of the timer being set.

Implementation:

If 'timeout' is not MESSYS__INFINITE, the function sets the timer clock going so that we get a timeout if there is no response within 'timeout' milliseconds. The ATIMER package is used to handle timers.

The function then looks for a message on any of this task's receive queues and returns the message components.

AMS_REPLY
Send a message on a given (path,messid)

Invocation:

```
(void)ams_reply( path, messid, message_function, message_status,
message_context, message_name, message_length, message_value, status )
```

Arguments:

path = int (given)
the path number for communicating with the other task

messid = int (given)
the number identifying the transaction

message_function = int (given)
message function

message_status = int (given)
message status

message_context = int (given)
message context

message_name = char * (given)
message name

message_length = int (given)
length of value

message_value = char * (given)
message value

status = int * (given and returned)
global status

Description:

As part of transaction 'messid' on path 'path', the user wishes to send the message AS A REPLY to a previously received message from the other end of the path. The user can ONLY reply with a MESSYS__DE_INIT message (something has gone wrong) or with a normal MESSY__MESSAGE message. Any other value of message_function will result in status MESSYS__MSGFUNC being returned.

A MESSYS__MESSAGE message may be a GSOC_END, terminating a transaction or one of the various other types, MESSYS__INFORM, MESSYS__PARAMREQ etc.

Implementation:

We first check that the path is open and that the transaction is in use. If both these are OK we check the 'function' part of the external form and check exactly what is being sent. If the process ends up sending a GSOC_END message the transaction is also closed at this end.

AMS_RESMSG
Send a RESCHEDULE message to this task

Invocation:

```
(void)ams_resmsg( length, value, status )
```

Arguments:

length = int (given)
number of significant bytes in value

value = char * (given)
message to be passed to main-line code

status = int * (given and returned)
global status

Description:
Causes the task to send specified RESCHED message 'value', qualified by 'name' to itself.

Implementation:
Send an OBEY message to the reschedule queue.

AMS_SEND
Send a message on a given path

Invocation:

```
(void)ams_send( path, message_function, message_status,
message_context, message_name, message_length, message_value, messid,
status )
```

Arguments:

path = int (given)
pointer to the path

message_function = int (given)
message function

message_status = int (given)
message status

message_context = int (given)
message context

message_name = char * (given)
message name

message_length = int (given)
length of value

message_value = char * (given)
message value

messid = int * (returned)
message number issued by this task (returned)

status = int * (given and returned)
global status

Description:

This function is used by the application code to send a message to another task on path 'path'.

The expectation is that this is one of a DEINIT message, an INIT message or the first message of a NEW transaction whose transaction number is to be set into '*messid'.

Implementation:

When used for the first message of a transaction, ams_getfreetrans() is used to obtain a free transaction and the MESSYS__MESSAGE is sent to the other task's command queue (using ams_sendgsocstart()).

When used to send an INIT message, ams_getfreetrans() is used to obtain a free temporary transaction and the MESSYS__INIT is sent to the other task's command queue (using ams_sendinit()).

When used to send a DEINIT, ams_senddeinit() is used to send the MESSYS__DE_INIT message to the other task's command queue.