ANGLO-AUSTRALIAN OBSERVATORY AAO/MESS_6
DRAMA Software Report 6
Version 0.4.1

Tony Farrell
23-Feb-95

A portable (VMS compatible) message code system

Contents

Revisions:

V0.4.1 23-Feb-1995 Update program and routine descriptions.

1 Introduction

The `MESSGEN` utility and the `Mess` routines provide a portable technique for generating unique error codes and then associating text with the error codes at run time.

`MESSGEN` will generate, if required, include files defining constants for each error code, in the C, Fortran, (Vax) Pascal [1] and TCL languages.

The `Mess` routines provide the ability to fetch the text associated with each message at program run time.

The grammer used to specify the message codes and associated text is a subset of that accepted by the VMS message utility and generates codes compatible with those generated by VMS message. This allows existing VMS code to be ported to other machines and allows existing user interfaces running on VMS machines to translate MESSGEN error codes into text.

2 Error handling

2.1 A survey of error handling techniques

Various techniques are used for handling and reporting errors within computer programs. Most of these are a combination of two basic handling techniques and two basic reporting techniques.

Normally, a routine/function encounting an error which it cannot recover from will do one of two things-

1. Return a value indicating that an error has occurred.
2. Cause an exception.

The first of these is simple but requires constant checking of status values, thus the second of these is often preferred. Unfortunately, to be done properly, it requires good operating system and language support. It is very hard to use in a multiple language environment in a portable manner.

The two possible reporting techniques are

1. Output details of the error in the routine in which the error has occurred
2. Return or set an error code which allows details of the error to be determined.

The first technique ensures that full details of errors are output (such as the name of files involved etc), but causes problems when higher level code wishes to handle the error itself.

The second technique provides good control over error reporting for the upper level software, but has trouble providing complete details of the error (such as the names of files involved).

[1] Since the original Pascal Standard does not support4 include files, its not clear how portable the Pascal output is.

2.1.1 The VMS technique

VAX/VMS uses a combination of both error handling techniques. Most VMS routines return a status code, although some cause exceptions. Error report is supported by the provision of a high quality message code system. It allows every routine to return a specific error code (although many return generic system wide codes) and provides a unique mapping of these error codes to messages.

User interface code simply looks up the unique text associated with each error code and outputs it as necessary using system calls.

The VMS exception system allows additional information to be added to the message, such as additional supplemental messages or the names of files.

Users may define their own message codes which can then be handled correctly by VMS. System managers can ensure such codes are unique by prescribing `Facility` numbers for each system.

There are two drawbacks. First, if the exception technique is not used, information (such as file names) can be lost. Second, it is VMS specific.

2.1.2 The UNIX technique

Under UNIX and most similar systems (e.g. VxWorks), the first error handling technique is used. Most routines return a value which indicates and error has occurred. This value is normally just something like zero or minus one. Additionally, the originator of the error sets a global variable (errno) to a value which indicates the reason for the error. Although there is generally a string associated with each errno value, it is not always appropriate or specific enough.

There is no way for programmers to associate text with their own codes without rebuilding the operating system. Additionally, there is no technique specified for generating unique error codes.

Generally, error reporting is done by higher level code writting messages.

2.1.3 The Starlink (ADAM) technique

Starlink software uses a complex technique designed for building software systems out of multiple layers of libraries.

Generally, each routine has a status argument. If the status argument is non-zero on entry, then the routine does nothing. If an error occurs in the routine, then, status is set to some value (normally a VMS message code, since Starlink software originated on VMS machines).

Additionally, any routine setting status is supposed to report an error. The routines for reporting an error allow the routine to save an appropriate text message (including file names etc., if necessary) in such a way that higher software levels can do one of the following-

- Add further error reports to provide more context.
- Write all reported messages to the user.
- Clear status and all the reported messages.

Starlink provides various techniques allowing for most possible cases. By making use of VMS message codes, unique status values can be used, allowing upper level software good control over error handling.

Unfortunately, a lot of starlink software just sets bad status, without reporting the error (Although this is slowly changing). This has generally not been too much of a problem as the VMS message system allowed messages to be associated with status values, but it causes problems with UNIX software.

Also, the error reporting software is a complex and large package in itself, not always suitable for inclusion in small real time tasks.

With the move to UNIX, Starlink is discouraging the use of VMS message codes, particularly the reliance on the association of a text string with the error code.

2.2 AAO real time system requirements

AAO VMS software has generally made extensive use of the VMS message code system. It works well in a real time system involving multiple processes, since message codes are easy to pass between processes. Because it is easy to generate unique message codes, we have not been inclined to use the Starlink error reporting system very much, although it is used to add context to errors.

With the move to UNIX systems, our reliance on VMS style error codes is a potential problem. Although we do intend to provide a Starlink style error reporting system (somewhat simplified), as mentioned above, it is not always suitable for real time systems, especially distributed ones. Additionally, we still intend using current VMS based user interfaces for many systems.

As a result, we find it desirable to be able to generate and make use of VMS style message codes on UNIX machines. The `MESSGEN` utility and `Mess` routines provide this ability.

3 Description

The messages generated this system are normally displayed to the user as a line of alphanumeric codes and text explaining the condition that caused the message to be issued.

Messages are normally displayed in the following format:

```
%FACILITY-L-IDENT, message-text
```

FACILITY
Specifies the abbreviated name of the software component that issued the message.

L
Show the severity level of the condition that caused the message. The five severity levels are represented by the following codes

S Success.

I Informational.

W Warning.

E Error.

F Fatal or severe.

IDENT
Identifies a symbol which represents the message

message-text
Explains the cause of the message.

4 Constructing Messages

You construct messages by writing a message source file (a .msg file) and compiling it with the `MESSGEN` utility.

For each message, a 32 bit code is generated. The exact bit order is dependent upon the machine byte order, but for a VAX, the format is

Bits 0-2 The severity, as follows

Symbol	Value	Description
STS_K_WARNING	0	Warning.
STS_K_SUCCESS	1	Success.
STS_K_ERROR	2	Error.
STS_K_INFO	3	Informational
STS_K_SEVERE	4	Severe (Fatal) error
	5	Reserved
	6	Reserved
	7	Reserved

These symbols are defined in `mess.h`, as are the symbols STS_M_SEVERITY (7) and STS_M_NOTSEVERITY (The inverse of severity).

Bits 3-14 Message number (1 - 4095).

Bit 15 Reserved, must be set true.

Bits 16-26 Facility (system) number (1 - 2047).

Bit 27 Reserved, must be set true.

Bits 28- 32 Reserved, must be set false.

You can refer to the message code in your programs by means of Symbols defined in include files. These include files (C, Fortran or Pascal) can be generated by `MESSGEN`. These symbols consist of

- The symbol prefix defined in the facility directive.
- The symbol name defined in the message definition.

4.1 The message source file

The message source file consists of message definition statements and directive that define the message text, the message code values and the message symbol. The various elements that are normally including in a message source file are-

- Facility directive.
- Severity directive.
- Base message number directive.
- End directive.
- Comments.

The first non-comment statement must be a `.FACILITY` directive. All messages defined after a `.FACILITY` statement are associated with that facility. A `.END` directive ends the list of messages associated with a particular facility. Currently, only one facility can be defined in a file.

Severity levels can be specified by a `.SEVERITY` directive or by including a severity qualifier as part of the message definition. (With `VMS MESSAGE`, you must specify the severity, but with `MESSGEN`, it defaults to `ERROR`).

The following is an example of a message file-

```
.FACILITY    DITS,2000/PREFIX=DITS__
!
! these lines are comments
!
.SEVERITY    FATAL
!
TASKDISC <Task disconnected>
MACHLOST <The machine on which the task was running has been lost>
FINDINGPATH <Already trying to find a path to this task>
INVMSGLEN <Message length is too small for a Dits message>/WARNING
.END
```

The facility number (2000), is what makes the message numbers unique. You should ensure you have chosen a message number unique to your system. [2]

[2]The AAO uses the facility numbers 1800 to 1899, allocated by starlink. A facility number can be allocated by editing the first message in the PROG bulletin folder.

4.2 Compiling the Message Source file

Message source files must be compiled before the messages defined in them can be used.

The `-c` option to the `messgen` command causes a C language include file to be generated. This include file will contain definitions for the symbols `DITS__TASKDISC`, `DITS__MACHLOST`, `DITS__FINDINGPATH` and `DITS__INVMSGLEN`. The `DITS__` part has come from the `PREFIX` specification.

The are various other options to chose from, see Appendix-A for more details, but of particular interest is the `-t` option, which generates the table file referenced in the next section.

Message source files can also be compiled with the VMS `MESSAGE` command to generate object files which can be linked with VMS programs, making the message text available to VMS system services, such as `SYS$GETMSG`.

5 The `Mess` routines

The `Mess` routines provide the ability, given the message code, to fetch the string specified in the message file for a message.

When the `-t` option is specified to the `messgen` command, a file with a suffix of `.msgt.h` (`_msgt.h` under VMS) is generated. This include file defines a table, named `MessFac_`*facname*, where *facname* is the facility name. In the above example, the table name is `MessFac_DITS`. This table specifies the message text for each message code in the facility.

For each message facility which may be used by a program, a call of the form-

```
MessPutFacility(&MessFac_DITS);
```

should be made, from a module which included the file generated by the `-t` option. Such a call makes the facility known to the `Mess` routines.

Normally, the initialisation routine of a package will call `MessPutFacility` for the message facility it uses.

To fetch the text associated with a message, the `MessGetMessage` routine is used. The following example uses the above message definition file-

```
#include "mess.h"
#include "dits.h"
#ifdef VMS
#    include "dits_msgt.h"
#else
#    include "dits.msgt.h"
#endif
#include <stdio.h>
```

```
int main()
{
    char buffer[200];
    MessPutFacility(&MessFac_DITS);

    MessGetMsg(DITS__TASKDISC,0,200,buffer);
    printf("%s\n",buffer);

}
```

When compiled and run, this example produces the output-

```
%DITS-F-TASKDISC, Task disconnected
```

The `MessPutFlags` routines allows the programmer to turn off output of various parts of the message text.

6 Compatibilty with VMS messages

The files accepted by `MESSGEN` are defined to be a subset of the files accepted by the `VMS MESSAGE` utility. It should be noted that although MESSGEN is defined in this way, some rescrictions imposed by `VMS MESSAGE` do not exist in `MESSGEN` and you can therefore create files accepted by `MESSGEN` but not by `VMS MESSAGE`.

6.1 VMS MESSAGE features which are not supported

The following `VMS MESSAGE` features are not supported by `MESSGEN`.

- `/SHARED` and `/SYSTEM` qualifiers to the `.FACILITY` directive. These qualifiers are `VMS` specific.
- Use of FAO arguments within messages. These require `VMS` specific system services are only of use with the `VMS` exception system.
- `/FAO_COUNT` qualifier to message definitions. Relevant to FAO arguments.
- `/USER_VALUE` qualifier to message definitions.
- Expressions.
- Use of the dollar sign ($) within message names. Most `UNIX` compilers do not accept them, so a warning is output by `MESSGEN`.
- You may only define one facility in a file.

6.2 VMS MESSAGE features which are ignored

The `IDENT, .LITERAL, .PAGE` and `.TITLE` directives are ignored, as is anything on the rest of the line they are on. (They are treated as comment initiators).

6.3 VMS MESSAGE restrictions lifted in MESSGEN

In MESGEN, identifier names can be of any length and the parameters to directives need not be on the same line as the directive.

A Programs

This section details the various programs available in the Mess systems.

A.1 messgen — Compile message definition files to produce various include files

Function: Compile message definition files to produce various include files

Synopsis:

```
messgen [-cofpdtl] [-C cname] [-F fname] [-P pname]
        filename[.msg]
```

Description: This program compiles a message definition file. It is capable of producing include files defining constants for message codes in The C, Fortran and Pascal languages. Additionally, it can produce a message table, which can be supplied in a call to MessPutFacility() to make the message details available to calls to MessGetMsg().

The C lanaguage include files are bracketed in an #ifdef statement such that they are only included once.

Options:

-c Generate a C language include file which defines constants of the form "PrefixName", for each message. The definitions are written to the file "basename.h".

-C cname As per -c, but the definitions are written to the file "basename.cname".

-f Generate a Fortran language include file which defines constants of the form "PrefixName", for each message. The definitions are written to the file "basename.f" on unix machines and the file "basename.for" on VMS machines

-F cname As per -f, but the definitions are written to the file "basename.fname".

-p Generate a Pascal language include file which defines constants of the form "PrefixName", for each message. The definitions are written to the file "basename.pin".

-P cname As per -p, but the definitions are written to the file "basename.pname".

-d Generate a TCL language include file which defines variables of the form "PrefixName", for each message. The definitions are written to the file "basename.tcl".

-D dname As per -p, but the definitions are written to the file "basename.dname".

-o When writing include files, generate the symbol "PrefixOK", with a value of 0.

-t Generate a message table which can be used when calling MessPutFacility. This is written to the file "basename_msgt.h"

-l Log the opening of input and output files.

-j Produce Java to the file "basename.java". Note, the class name will be "basename". The resulting class extends DramaStatus.

-J package Produce java, but to be within the specified package. File is "basename".java. Note, the class name will be "basename". The resulting class extends DramaStatus.

-x filename Specifies the output file name explicity. If used, only one output type can be selected.

basename is part after the last slash if any, minus the suffix ".msg" if it exists.

See Also: MessPutFacility(), MessGetMsg(), Vax/VMS message utility manual.

Support: Tony Farrell, AAO

A.2 messana — Analyze message codes.

Function: Analyze message codes.

Synopsis: messana [code...]

Description: Each message code (which should be an integer in a form understood by the C RTL function strtol()) is analyzed and its facility number, message number and severity output.

Support: Tony Farrell, AAO

B Mess routine descriptions

This appendix describes the various Mess routines.

B.1 MessFacility — Return the facility number associated with a message code.

Function: Return the facility number associated with a message code.

Description: Analyzes a mesage code and returns the facility assocaited with the message.

Language: C

Call:
(int) = MessFacility (code)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) code (StatusType) The message code.

Include files: mess.h

Support: Tony Farrell, `AAO`

B.2 MessFacilityKnown — Returns true if the specified facility is known to the system.

Function: Returns true if the specified facility is known to the system.

Description: This routine returns true if the specified facility is known to the system.

Language: C

Call:
(int) = MessFacilityKnown (int facilityNum)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(int) facilityNum (int) The number of the facility to check for.

Include files: mess.h

External functions used: None

External values used: None

Prior requirements: None

Support: Tony Farrell, AAO

B.3 MessGetMsg — Return message text for a given message number.

Function: Return message text for a given message number.

Description: Searches the known message facilities for a message of the given number. In this search, the severity of the supplied message number is ignored. If found, a message string is constructed according to the details requested in the flags argument. The format is-

"`%FACILITY-S-NAME`, `text`"

where-

`FACILITY`	The name of the correspond facility.
S	The severity of the message, as supplied in the call to this routine (The same message can have different severities).
`NAME`	The message name.
text	The text associated with the message.

If the message cannot be found, a suitable default text is returned. As a special case, the message number 0 is always defined and returns an appropriate `OK` text.

When running under `VMS` the behaviour changes. If a message cannot be found, then the `SYS$GETMSG` system service is called to attempt to find the message in the `VMS` tables. This will work if the message is a `VMS` system message or the message is known to the current process (either an object file generated by message/object is linked to the current image or a "set message" command has been used to add the message to the current process)

Language: C

Call:

(int) = MessGetMsg (msgid, flags, buflen, buffer)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) msgid (StatusType) The identification of the message to be retrieved.

(>) flags (Int) Message components to be returned. Set the following bits true to include the component -

`MESS_M_TEXT`	Message text.
`MESS_M_IDENT`	Message identifier.
`MESS_M_SEVERITY`	Message severity indicator.
`MESS_M_FACILITY`	Facility name.

If flags is 0, then the default flags, set by MessSetFlags, is used. If MessSetFlags has not been called, then all are set true, except under `VMS`, where the processes `VMS` message mask is used.

(>) buflen (Int *) The length of buffer.

(>) buffer (Char *) The buffer to place the message in. If it is too long, then the message is truncated.

Include files: mess.h

External functions used:

strncat	CRTL	Concentrate one string to another.
sprintf	CRTL	Formated print to a string.
lib$getjpi	VMS-RTL	(VMS version only) Get job process information.
sys$getmsg	VMS	(VMS version only) Get VMS message text.

Function value: Return 1 if the message is found and 0 if it is not.

External values used: None

Prior requirements: None

Support: Tony Farrell, AAO

B.4 MessNumber — Return the number of a message

Function: Return the number of a message

Description: Analyzes a message code and returns the number of the message within the facility

Language: C

Call:
(int) = MessNumber(code)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) code (StatusType) The message code.

Include files: mess.h

Support: Tony Farrell, AAO

B.5 MessPutFacility — Add a new message facility to the list of known facilities

Function: Add a new message facility to the list of known facilities

Description: The new message facility is added to the beginning of the list. Any other facility of the same number will be ignored.

Under `UNIX` and `VMS`, this list is kept on a process specific basis. Under VxWorks, it is kept on a global basis.

Under VxWorks, a mutual exclusion semaphore is used to protect the data structure against two or more tasks accessing the list at the same time. If the semaphore has already been taken by a task, other tasks will block until it becomes available. Since the Semaphore is priority inversion safe, task deletion safe, queues pended tasks on the bases of their priority and the operation is quick, no task should block for any length of time. This makes the use of this routine safe in all cases, except from interrupt handlers, thus, this routine should not be called from interrupt handlers.

Language: C

Call:

(Void) = MessPutFacility (facility)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) facility (MessFacType *) The new facility as generated by the messgen utility.

Include files: mess.h

External functions used:

semMCreate	VxWorks	VxWorks only, create a mutual-exclusion semaphore.
semTake	VxWorks	VxWorks only, take a semaphore.
semGive	VxWorks	VxWorks only, give a semaphore.

External values used: None

Prior requirements: None

Support: Tony Farrell, `AAO`

B.6 MessPutFlags — Sets the default components when a message is returned.

Function: Sets the default components when a message is returned.

Description: This routine only has effect on `MessGetMsg` if that routine is called with its `flags` argument equal to 0.

If this routine is not called, then all components of a message are returned by `MessGetMsg`, otherwise this routine sets the components to be returned.

An exception is under `VMS`. Under `VMS`, if this routine is not called, `MessGetMsg` uses the default `VMS` message flags.

Language: C

Call:
(Void) = MessPutFlags (int flags)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

(>) flags (Int) Message components to be returned. If flags is specified as 0, then components are return. Set the following bits true for include the component

`MESS_M_TEXT`	Message text.
`MESS_M_IDENT`	Message identifier.
`MESS_M_SEVERITY`	Message severity indicator.
`MESS_M_FACILITY`	Facility name.

Include files: mess.h

External functions used: None

External values used: None

Prior requirements: None

Support: Tony Farrell, `AAO`

B.7 MessSeverity — Return the severity of a message

Function: Return the severity of a message

Description: Analyzes a message code and returns the severity of the message

Language: C

Call:
(int) = MessSeverity(code)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

(>) code (StatusType) The message code.

Include files: mess.h

Support: Tony Farrell, AAO

B.8 MessStatusToSysExit — Given a status code, return an exit value.

Function: Given a status code, return an exit value.

Description: This routine returns an appropiate value to be used with the main() function return statement, given a status code.

Language: C

Call:
(int) = MessStatusToSysExit (StatusType code)

Parameters: ("**>**" input, "**!**" modified, "W" workspace, "**<**" output)

(>) code (StatusType) The message code to convert.

Include files: mess.h

Prior requirements: None

Support: Tony Farrell, AAO

C Source File Statements

C.1 Base Message Number Directive

Defines the value used in constructing the message code.

FORMAT	.BASE number
statement parameter	number Specifies a number to be assocaited with the next message definition
statement qualifiers	none

Description

By default, all of the message following a facility directive are numbered sequentially, beginning with 1.

If you want to supersede this default numbering system, (for example, if you want to reserve some message numbers for future assignment) specify a message number of you choice using the base number directive. The message number is used as a base for the sequential numbering of all messages that follow until another `.BASE` or the `.END` is encounted.

Example

```
.FACILITY    SAMPLE,1/PREFIX=ABC_A
.SERVERITY   ERROR

UNRECOG      < Unrecognised keyword>
AMBIG        < Ambiguous keyword>

.SEVERITY    WARNING
.BASE        10
SYNTAX       < Invalid syntax>
.END
```

The the first two message numbers are defined as 1 and 2. This sequential numbering is superseded by the base message message number directive, which assigns the message number 10 to the third message.

C.2 End Directive

Terminates the entire list of messages for the facility.

FORMAT	.END
statement parameter	none
statement qualifiers	none

Description

An End directive terminates the entire list of messages for a facility.

Example

```
.FACILITY    SAMPLE,1/PREFIX=ABC_A
.SERVERITY   ERROR

UNRECOG      < Unrecognised keyword>
AMBIG        < Ambiguous keyword>

.SEVERITY    WARNING
.BASE        10
SYNTAX       < Invalid syntax>
.END
```

The .END directive terminates the list of messages for the **SAMPLE** facility.

C.3 Facility Directive

Specifies the facility to which messages will apply.

FORMAT	.FACILITY facnum[,]facnum/qualifier
statement parameter	facnam Specifies the facility name used in the facility field of the message and in the symbol representing the facility number. facnum Specifies the facility number that is used to construct the 32-bit value of the message code. A decimal value in the range 1 to 2047.
statement qualifiers	/PREFIX=prefix defines an alternate symbol prefix to be used in the message symbol for all messages referring to this facility. The default prefix is the facility name, followed by an underscore (_).

Description

The facility directive is the first directive in the message file. Both the facility name and the facility number are required and can be separated by a comma or by any number of spaces or tabs.

Example

```
.FACILITY    SAMPLE,1/PREFIX=ABC__
.SERVERITY   ERROR

UNRECOG      < Unrecognised keyword>
AMBIG        < Ambiguous keyword>

.SEVERITY    WARNING
.BASE        10
SYNTAX       < Invalid syntax>
.END
```

The facility statement in this message source file defines the messages belonging to the `SAMPLE` facility with facility number 1. The `/PREFIX=ABC__` qualifier defines the message symbols `ABC__UNRECOG`, `ABC__AMBIG` and `ABC__SYNTAX`.

C.4 Message definition

Defines the message symbol and the message text.

FORMAT	name < message-text > /qualifier
statement parameters	**name** Specifies the name that is combined with the symbol prefix (defined in the facility directive) to form the message symbol. The name is used in the `IDENT` field of the message. **message-text** Defines the text explaining the condition that caused the message to be issued. The message text can be delimited either by angle brackets or by quotation marks. The text can be of any length, but cannot be continued onto another line.
statement qualifiers	**/SUCCESS** Specifies the success level for a message. This qualifier overrides any `.SEVERITY` directive in effect. **/INFORMATIONAL** Specifies the informational level for a message. This qualifier overrides any `.SEVERITY` directive in effect. **WARNING/** Specifies the warning level for a message. This qualifier overrides any `.SEVERITY` directive in effect. **ERROR** Specifies the error level for a message. This qualifier overrides any `.SEVERITY` directive in effect. **SEVERE** Specifies the severe level for a message. This qualifier overrides any `.SEVERITY` directive in effect. **FATAL** Specifies the fatal level for a message. This qualifier overrides any `.SEVERITY` directive in effect. Fatal and severe are equivalent.

Description

The message definition specifies the message text that will be displayed and the name used in the `IDENT` field of the message. Additionally, you can use the message definition to specify the severity level for the message.

Example

```
.FACILITY    SAMPLE,1/PREFIX=ABC__
.SERVERITY   ERROR

UNRECOG      < Unrecognised keyword>
AMBIG        " Ambiguous keyword"

.SEVERITY    WARNING
.BASE        10
SYNTAX       < Invalid syntax>
.END
```

This message source file contains a facility directive and three message definitions.

C.5 Severity Directive

Specifies the severity level to be associated with the messages that follow.

FORMAT	.SEVRITY level
statement parameter	level Specifies the level of the condition that caused the message. SUCCESS — Produces a S code in a message. INFORMATIONAL — Produces an I code in a message. WARNING — Produces a W code in a message. ERROR — Produces an E code in a message. SEVERE — Produces a F code in a message. FATAL — Produces a F code in a message. SEVERE is equivalent to FATAL and they can be used interchangeable; the severity level code for both of these if `F`.
statement qualifiers	none

Description

Following the facility directive, the message source file generally contains a severity directive.

Example

```
.FACILITY    SAMPLE,1/PREFIX=ABC__
.SERVERITY   ERROR

UNRECOG      < Unrecognised keyword>
AMBIG        < Ambiguous keyword>

.SEVERITY    WARNING
.BASE        10
SYNTAX       < Invalid syntax>
.END
```

The two severity directives include in this message source defines the severity levels for three messages. The first two messages have a severity level of `E`; the third message has a severity level of `W`.