10

Message APIs

Messages are a way of life with i5/OS. Every time you start, hold, or cancel a job, the system issues a message. While your job is executing, the system sends messages to the job log. If the system needs human intervention, such as pressing the Start button on a printer or loading a tape on a tape drive, the system notifies the interested parties by sending a message.

Since messages are such an integral part of i5/OS, you would probably like to know when certain ones are sent on the system. For this, the system provides an exit capability through *watches*. Watches allow you to define a program that should be automatically called when a particular message is sent to the QSYSOPR system-operator message queue, the QHST history log, the job log of one or more jobs, or a regular message queue. This chapter first looks at how to write watch exit programs and then how to associate a watch exit program with messages that might be sent on the system.

Application programs often want to send messages to either another program or a user. The system provides for many types of messages to meet the various needs of developers. These messages include completion messages to indicate that a particular piece of work has finished, status messages to indicate the current status of work, diagnostic messages to provide possible error-related information, and escape messages to indicate a general

failure. This chapter also looks at how to send messages from your application program using the Send Program Message API, QMHSNDPM.

Messages are frequently stored as message descriptions in a message file. The chapter concludes with a look at how the Retrieve Message API, QMHRTVM, can be used to search for messages in a message file that have a particular word in either its first- or second-level text. Many times, we've wondered if there was already a message describing a certain situation, but had no fast, easy way to determine what that message might be. With this API, that problem has been solved!

Watch Exit Programs

Watches provide the ability to have i5/OS call one or more user programs when a particular message is sent to the system operator, the history log, the job log of one or more jobs, or to a message queue. There is the Start Watch API (QSCSWCH) to begin a watch session, the End Watch API (QSCEWCH) to end a watch session, and the Watch for Event exit program capability. All three are discussed in the "Problem Management" category of System APIs under the subcategory "Monitoring."

We'll start by explaining how to write a watch exit program. Following that, you'll see how to use the Start Watch API to associate a watch exit program with messages and message queues on the system. The parameter descriptions for a watch exit program are shown in Table 10.1.

Table 10.1: Watch Exit Program Parameters

Parameter	Description	Type	Size
1	Watch Option Setting	Input	Char(10)
2	Session ID	Input	Char(10)
3	Error Detected	Output	Char(10)
4	Event Data	Input	Char(*)

The first parameter, Watch Option Setting, tells the exit program why it was called. Watch exit programs can currently be called in two situations. One is if a watched-for message is sent on the system (*MSGID), and the other is if a watched-for Licensed Internal Code (LIC) error log entry is recorded on the system (*LICLOG). The two values

supported, therefore, are *MSGID and *LICLOG. Although you can start a watch for either of these situations, we only discuss message watches in this book.

The second parameter, Session ID, is the name of the session calling the exit program. When a watch is started, a session ID can either be generated by the system or specified by the user.

The third parameter, Error Detected, is an output from the exit program back to i5/OS. If the exit program returns blanks for this parameter, no error was encountered in the exit program, and i5/OS should continue calling the exit program whenever the watched-for message is sent again. Any other value informs i5/OS that an error has been encountered within the exit program, and i5/OS will stop calling the exit program.

The fourth parameter, Event Data, is data associated with the message that was sent and that triggered the call to the watch exit program. For messages (a watch session setting of *MSGID), the format of this event data is shown in Figure 10.1.

```
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++
D*****************************************************************
D*Watch Exit Program called because a message id and any
D*associated comparison data is matched.
D*This structure is for the user exit program called by
D*STRWCH cmd or Start Watch (QSCSWCH) API
D*****************************************************************
DESCQWFM          DS
D*                                              Qsc Watch For Msg
D ESCLWI                  1      4B 0
D*                                              Length Watch Information
D ESCMID00                5     11
D*                                              Message ID
D ESCERVED01             12     12
D*                                              Reserved
D ESCMQN                 13     22
D*                                              Message Queue Name
D ESCMQL                 23     32
D*                                              Message Queue Lib
D ESCJN                  33     42
D*                                              Job Name
D ESCUN                  43     52
D*                                              User Name
```

Figure 10.1: A watch message format from QSYSINC/QRPGLESRC member ESCWCHT (part 1 of 3).

```
D ESCJNBR                 53     58
D*                                          Job Number
D ESCRSV2                 59     62
D*                                          Reserved2
D ESCSPGMN                63    318
D*                                          Sending Program Name
D ESCSPGMM               319    328
D*                                          Sending Program Module
D ESCOSP                 329    332B 0
D*                                          Offset Sending Procedure
D ESCLOSP                333    336B 0
D*                                          Length Of Sending Proced
D ESCRPGMN               337    346
D*                                          Receiving Program Name
D ESCRPGMM               347    356
D*                                          Receiving Program Module
D ESCORP                 357    360B 0
D*                                          Offset Receiving Procedu
D ESCLORP                361    364B 0
D*                                          Length Of Receiving Proc
D ESCMS                  365    368B 0
D*                                          Msg Severity
D ESCMT                  369    378
D*                                          Msg Type
D ESCMTOO                379    386
D*                                          Msg Timestamp
D ESCMK                  387    390
D*                                          Msg Key
D ESCMFILN               391    400
D*                                          Msg File Name
D ESCMFILL               401    410
D*                                          Msg File Library
D ESCRSV3                411    412
D*                                          Reserved3
D ESCOCD01               413    416B 0
D*                                          Offset Comparison Data
D ESCLOCD01              417    420B 0
D*                                          Length Of Comparison Dat
D ESCCA                  421    430
D*                                          Compare Against
D ESCRSV4                431    432
D*                                          Reserved4
D ESCCCSID               433    436B 0
D*                                          Comparison Data CCSID
D ESCOCDF                437    440B 0
D*                                          Offset Comparison Data F
```

Figure 10.1: A watch message format from QSYSINC/QRPGLESRC member ESCWCHT (part 2 of 3).

```
D ESCORD                441    444B 0
D*                                      Offset Replacement Data
D ESCLORD               445    448B 0
D*                                     Length Of Replacement Da
D ESCCCSID00            449    452B 0
D*                                      Replacement Data CCSID
D*ESCSP                 453    453
D*
D*                                 variable length data @B2M
D*ESCRP                 454    454
D*
D*                                 variable length data @B2M
D*ESCCD01               455    455
D*
D*                                 variable length data @B2M
D*ESCRD                 456    456
D*
D*                                 variable length data @B2M
```

Figure 10.1: A watch message format from QSYSINC/QRPGLESRC member ESCWCHT (part 3 of 3).

As you can see, a wealth of information about the message is available to the exit program. This includes the message ID, the qualified message queue name, the name of the job sending the message, the program sending the message, the message severity, the message key, the replacement data associated with the message occurrence, and much more. (It should be pointed out that the exit program does not run in either the job sending the message or the job receiving the message—it runs in its own job.) With this amount of information, our exit program can be used in a variety of ways. One possible application is to automate various tasks on the system.

For example, when the history log of i5/OS becomes full, the messages are written to a database file. Message CPF2456 is then sent to the QSYSOPR message queue to inform the operator that the file is full and should be saved. Figure 10.2 shows a watch exit program that can automate the saving of new QHST files to save files.

Two specific fields within the QSYSINC-provided data structure ESCCQWFM are used in the example program. The first one, ESCMID00, contains the message ID that caused the watch program to be called. The other field, ESCORD, contains an offset to the replacement data associated with the message ESCMID00.

```
h dftactgrp(*no)

DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++
d/copy qsysinc/qrpglesrc,escwcht

dFig10_2          pr
d Type                          10    const
d Session                       10    const
d Error                         10
d Data                                likeds(ESCQWFM)

dFig10_2          pi
d Type                          10    const
d Session                       10    const
d Error                         10
d Data                                likeds(ESCQWFM)

dCmdExc           pr                  extpgm('QCMDEXC')
d Command                    65535    const
d CmdLength                     15  5 const
d IGC                            3    const options(*nopass)

dMsgDtaPtr        s                *
dMsgDta           ds              75    based(MsgDtaPtr)
d File                  1       10
d Lib                  11       20

dCommand          s               80
dWait             s                1

CL0N01Factor1+++++++Opcode&ExtFactor2+++++++Result++++++++Len++D+HiLoEq
 /free
  Error = *blanks;  // assume all is well until shown otherwise

  // Check to make sure this is a *MSGID watch
  if (Type = '*MSGID');

     // Check to make sure this is for CPF2456
     if (Data.ESCMID00 = 'CPF2456');

        //  Get to the message replacement data
        MsgDtaPtr = %addr(Data) + Data.ESCORD;
```

Figure 10.2: Watch for the message CPF2456—"Log version &1 in &2 closed and should be saved" (part 1 of 2).

```
        // Save the log
        Command = 'SAVOBJ OBJ(' + File + ') LIB(' + Lib +
                          ') DEV(*SAVF) SAVF(QGPL/MYSAVES)';
        CmdExc( Command :%len(%trimr(Command)));

      // Not CPF2456
      else;
         dsply ('Unexpected ' + Data.ESCMID00 + ' received.') ' ' Wait;
          Error = '*ERROR';
      endif;

   // Not a *MSGID watch
   else;
      dsply ('Wrong type ' + %trimr(Type) + ' received.') ' ' Wait;
      Error = '*ERROR';
   endif;

   *inlr = *on;
   return;
  /end-free
```

Figure 10.2: Watch for the message CPF2456—"Log version &1 in &2 closed and should be saved" (part 2 of 2).

After initializing the Error parameter to blanks (indicating the exit program has no errors and should continue to be called by i5/OS), the program checks to see if it is being called for a message watch (Type = '*MSGID'). If not, the appropriate error text is DSPLYed and the Error parameter is set to *ERROR so that the watch exit program is not called again. If Type is *MSGID, the program checks to see if it's being called for message ID CPF2456 (Data.ESCMID00 = 'CPF2456'). If not, appropriate error text is DSPLYed and the Error parameter is again set to *ERROR.

If both checks are successful, the program accesses the message-replacement data by setting the pointer MsgDtaPtr to the address of the Data parameter and adding to this address the offset to the replacement data (Data.ESCORD). It does this because message CPF2456 returns the file name and library name of the new QHST file in the first 20 bytes of the message replacement data, and we have plans for that information. We know how this replacement data is defined by displaying the message description CPF2456 in QCPFMSG and looking at the message-replacement variable usage and definitions. The data structure MsgDta defines the two fields, File and Lib, that map to the replacement data we are interested in. As MsgDta is BASED on MsgDtaPtr, the program now has direct access to these values.

The program then constructs a SAVOBJ command string, using the watched message replacement data (fields File and Lib) to set the OBJ and LIB parameters of the SAVOBJ command. The program also specifies that the save is to save file QGPL/MYSAVES. Note that we have made a simplifying assumption here, namely, that the save file is empty or previously cleared. In a production environment, you would probably want dynamic creation of the save file and the usage of appropriately named save files.

After the SAVOBJ command string has been generated, the program uses the Execute Command API, QCMDEXC (introduced in chapter 4) to run the command. The program then exits.

Starting a Watch

Having looked at the FIG10_2 program, let's now look at the Start Watch API, QSCSWCH. It is used to associate program FIG10_2 with the sending of message CPF2456 to the system operator message queue. The parameter descriptions for QSCSWCH are listed in Table 10.2.

Table 10.2: The Start Watch API (QSCSWCH)

Parameter	Description	Type	Size
1	Session ID	Input	Char(10)
2	Started Session ID	Output	Char(10)
3	Watch Program	Input	Char(20)
4	Watch for Message	Input	Char(*)
5	Watch for LIC Log Entry	Input	Char(*)
6	Error Code	I/O	Char(*)

The first parameter, Session ID, allows you to name the watch. This name will be displayed on commands such as Work with Watches (WRKWCH), used on APIs such as End Watch (QSCEWCH), and passed to the watch exit program at run time. A meaningful name can help both operators and developers. If you don't care to name a watch session, you can use the special value *GEN, and the system will generate a watch-session ID name for you.

The second parameter, Started Session ID, is an output from the API that is used to return the name of the started session. This is primarily used as feedback from the API when *GEN is used for the first parameter.

The third parameter, Watch Program, is the qualified name of the exit program to be called when the watched-for message is sent to a watched message queue. As with most qualified names used in API calls, the first 10 bytes is the name of the program, and the second 10 bytes is the name of the library where the program is located. This program will be called once each time the watched-for message is sent. If the watched-for message is sent to multiple message queues, the exit program will be called once for each occurrence of the message being sent to each of the watched message queues.

The fourth parameter, Watch for Message, is a structure that provides information to the system on what message(s) and queue(s) are to be watched. This structure starts with a Binary(4) field indicating how many messages are to be watched for. Immediately following this field is an array of variable-length sub-structures of the type shown in Figure 10.3. These sub-structures are where you define the messages to watch for.

The fifth, Watch for LIC Log Entry, is a structure that provides information to the system on what LIC log-entry identifiers are to be watched. The fifth parameter, Error Code, is the standard API error-code structure.

```
DName++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++
D*****************************************************************
D*Format for message information
D*****************************************************************
DQSCWFMF          DS
D*                                          Qsc Watch For Msg Fmt
D QSCLMI                  1      4B 0
D*                                          Length Message Informati
D QSCMID                  5     11
D*                                          Message ID
D QSCERVED               12     12
D*                                          Reserved
D QSCMQNO2               13     22
D*                                          Message Queue Name
D QSCMQL                 23     32
D*                                          Message Queue Lib
D QSCJN                  33     42
D*                                          Job Name
D QSCUN                  43     52
D*                                          User Name
D QSCJNBR                53     58
D*                                          Job Number
```

Figure 10.3: The format from QSYSINC/QRPGLESRC member QSCSWCH for starting a watch (part 1 of 2).

```
D QSCRSV2                 59     64
D*                                          Reserved2
D QSCOCD                  65     68B 0
D*                                          Offset Comparison Data
D QSCLOCD                 69     72B 0
D*                                          Length Of Comparison Dat
D QSCCA                   73     82
D*                                          Compare Against
D*QSCCD                   83     83
D*
D*                                    variable length data
```

Figure 10.3: The format from QSYSINC/QRPGLESRC member QSCSWCH for starting a watch (part 2 of 2).

There are several fields defined within the QSYSINC-provided QSCWFMF data structure. The first, QSCLMI, is set to the size of the current occurrence of the QSCWFMF data structure. This size then provides a displacement for the API to the next occurrence of the QSCWFMF data structure. The number of occurrences, or array elements, is controlled by the initial Binary(4) field provided at the start of the fourth parameter.

The QSCWFMF data structure shown in Figure 10.3 allows you to specify what message to watch for and what message queues to monitor for the watched message. Each structure can specify one message ID (QSCMID), and you can have up to 100 of these structures per call to the Start Watch API.

You can specify what queue to monitor for each message. The Message Queue Name field (QSCMQN02) can be a message queue name or a special value. The defined special values are *SYSOPR for the QSYSOPR message queue, *JOBLOG for watching the job log of one or more jobs, and *HSTLOG for the QHST message queue.

If you use the special value *JOBLOG for the message queue, you can also specify what jobs to watch. The Job Name Field (QSCJN) can be a specific name such as PLANT0001, a generic name such as PLANT*, or one of two special values. The special values are the single asterisk (*) for the current job, and *ALL for all jobs on the system. Similar to the Job Name field, the Job User Name (QSCUN) can be a specific name such as ROBERTS, a generic name such as ROB*, or the special value *ALL. The Job Number field (QSCJNBR) likewise supports a special value of *ALL.

You can also specify message-comparison data for each message. This allows you to filter which occurrences of a message actually cause the watch exit program to be called.

You might, for instance, only want the exit program called when a particular watched message is sent by program PGM01A, or when the message replacement data for a message contains a particular value (perhaps a specific object name). You can specify this comparison data by setting the fields QSCOCD, the offset to the comparison data, and QSCLOD, the length of the supplied comparison data, to appropriate values and then providing the comparison data at offset QSCOCD.

As you can see, this structure provides a lot of control over determining when you want your exit program to receive control. Figure 10.4 shows how to use the Start Watch API, QSCSWCH, to associate program FIG10_2 with message CPF2456 being sent to the system-operator message queue QSYSOPR.

```
h dftactgrp(*no)

DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++
d/copy qsysinc/qrpglesrc,qscswch
d/copy qsysinc/qrpglesrc,qusec

dStartWatch       pr                  extpgm('QSCSWCH')
d SessionID                     10    const
d StrSsnID                      10
d WatchPgm                      20    const
d Messages                   65535    const options(*varsize)
d LICs                       65535    const options(*varsize)
d QUSEC                               likeds(QUSEC)

dWatchPgm         ds
d Pgm                           10    inz('FIG10_2')
d Lib                           10    inz('*CURLIB')

dMessages         ds                  qualified
d NbrMsgs                       10i 0
d MsgFormat                           likeds(QSCWFMF)

dLICs             ds                  qualified
d NbrLICs                       10i 0

dStrSsnID         s             10
```

Figure 10.4: Start a watch for message CPF2456—"Log version &1 in &2 closed and should be saved" (part 1 of 2).

```
CL0N01Factor1+++++++Opcode&ExtFactor2+++++++Result++++++++Len++D+HiLoEp
 /free
  QUSBPRV = 0;

  // Watch for CPF2456 in the QSYSOPR message queue
  Messages.NbrMsgs = 1;
  Messages.MsgFormat = *loval;
  Messages.MsgFormat.QSCLMI = %size(Messages.MsgFormat);
  Messages.MsgFormat.QSCMID = 'CPF2456';
  Messages.MsgFormat.QSCMQN02 = '*SYSOPR';
  Messages.MsgFormat.QSCJN = *blanks;
  Messages.MsgFormat.QSCUN = *blanks;
  Messages.MsgFormat.QSCJNBR = *blanks;
  Messages.MsgFormat.QSCCA = *blanks;

  // No LIC logs are being watched for
  LICs.NbrLICs = 0;

  StartWatch('QHSTSAVES' :StrSsnID :WatchPgm :Messages :LICs :QUSEC);

  *inlr = *on;
  return;
 /end-free
```

Figure 10.4: Start a watch for message CPF2456—"Log version &1 in &2 closed and should be saved" (part 2 of 2).

Figure 10.4 defines a data structure, Messages, that has two sub-elements. The first is a Binary(4) field for the number of messages to be watched, NbrMsgs, and then one occurrence, MsgFormat, of the QSCWFMF structure defined in Figure 10.3.

After initializing the Error Code Bytes Provided field, QUSBPRV, to zero to indicate we want exceptions returned as messages, the program sets Messages.NbrMsgs to one and initializes the Messages.MsgFormat sub-structure. Messages.MsgFormat is first set to all null values (x'00'), as there are reserved fields within the structure. Then, specific subfields are set to their appropriate values. The length of the message information, Messages.MsgFormat.QSCLMI, is set to the size of Messages.MsgFormat, the message ID is set to CPF2456, the message queue is set to the special value *QSYSOPR, the full watched job name is set to blanks, and the Compare Against field is also set to blanks. The program then initializes the number of LIC log entries to watch, LICs.NbrLICs, to zero, as we're not interested in LIC log entries.

After this setup is complete, FIG10_4 calls the Start Watch API with a session ID name of QHSTSAVES, and ends. The watch is now active and, in this scenario, we have now automated the saving of QHST history files.

To end the watch session, use the Work with Watches command, WRKWCH WCH(*STRWCH). Select option 2, End, with the QHSTSAVES session.

Sending Program Messages

Applications often need to send messages to end users informing them of what is going on. This is especially true if the application is performing a long-running task. For this reason, i5/OS has the concept of status messages. Status messages typically show up at the bottom of the screen while a function is running, and then disappear. You will now learn how to send a status message from your application program.

Let's start by creating a message file named MSGS in the SOMELIB library. A user message file is not actually required to send messages, but using message files and message descriptions is a better way to write applications than using text embedded within an application. If you've ever been involved with an application program that needs to support translations into multiple national languages, you'll know this all too well! Using message files is the better way to write an application, so that's how we'll build this example. Figure 10.5 shows the command to create the message file and add the message description MSG0001 to the MSGS message file.

```
CRTMSGF MSGF(SOMELIB/MSGS)

ADDMSGD MSGID(MSG0001) MSGF(SOMELIB/MSGS) MSG('Job is doing some work.
Please be patient')
```

Figure 10.5: Create the message file SOMELIB/MSGS.

To send this message to the user, you use the Send Program Message API, QMHSNDPM. The parameter descriptions for this API are shown in Table 10.3. The full documentation for the API can be found in the "Message Handling" category of System APIs.

Do not be dismayed by the number of parameters and the references to call stack entries. In general, using this API is straightforward enough, but there is a lot of flexibility built into it for those who need direct control when sending messages from one program to another.

Table 10.3: The Send Program Message API (QMHSNDPM)			
Parameter	**Description**	**Type**	**Size**
1	Message Identifier	Input	Char(7)
2	Qualified Message File Name	Input	Char(20)
3	Message Data or Immediate Text	Input	Char(*)
4	Length of Message Data or Immediate Text	Input	Binary(4)
5	Message Type	Input	Char(10)
6	Call Stack Entry	Input	Char(*) or Pointer
7	Call Stack Counter	Input	Binary(4)
8	Message Key	Output	Char(4)
9	Error Code	I/O	Char(*)
Optional Parameter Group 1			
10	Length of Call Stack Entry	Input	Binary(4)
11	Call Stack Entry Qualification	Input	Char(20)
12	Display Program Messages Screen Wait Time	Input	Binary(4)
Optional Parameter Group 2			
13	Call Stack Entry Data Type	Input	Char(10)
14	Coded Character Set Identifier	Input	Binary(4)

The first parameter, Message Identifier, identifies what message you want to send. A message ID must be provided if you are sending an escape, notify, or status message. If you are sending an impromptu message, also known as *immediate text*, you can use blanks for this parameter.

The second parameter, Qualified Message File Name, specifies what message file is to be used to locate the message identifier passed in the first parameter. As with any API qualified name, the first 10 bytes are the message file name, and the second 10 bytes are the library where the message file can be found. The special values *CURLIB and *LIBL can be used for the library portion of this parameter.

The third parameter, Message Data or Immediate Text, can contain the message replacement data to be used with a predefined message's substitution variables (if a message ID was used for the first parameter), or the immediate text that is to be sent (if the first parameter was set to blanks).

The fourth parameter, Length of Message Data or Immediate Text, is the length of the data provided in the third parameter.

The fifth parameter, Message Type, specifies what type of message is being sent. The supported types are as follows:

- *CMD—Command
- *COMP—Completion
- *DIAG—Diagnostic
- *ESCAPE—Escape
- *INFO—Informational
- *INQ—Inquiry
- *NOTIFY—Notify
- *RQS—Request
- *STATUS—Status

For a complete definition of these types (and other types not supported by this API) see the Information Center.

The sixth parameter, Call Stack Entry, provides a reference point to an active program or procedure within your job that you want to send the message to. Special values include an asterisk (*) for the current call stack entry (that is, yourself) and *EXT for the external message queue of the job. There are several other special values for this parameter, but these two will suffice for our discussion of this API.

The seventh parameter, Call Stack Counter, provides the location within your job's call stack where the message should be sent. If a value of zero is used, the message is sent to the call stack entry identified with the sixth parameter. If a value of one is used, the message is sent to Call Stack Entry 1 before the call stack entry identified with the sixth

parameter. If a value of two is used, the message is sent to Call Stack Entry 2 before the call stack entry identified with the sixth parameter, and so on. If the sixth parameter is set to the special value *EXT, this parameter is ignored.

The eighth parameter, Message Key, is an output parameter. It can be used with other APIs to reference the message you are sending. The ninth parameter is the standard error-code structure found with many APIs. The remaining parameters are not used by any of the examples in this chapter, and so they are not discussed here. See the Information Center API documentation if you are interested in learning about these optional parameters.

The program to actually send the message MSG0001 created in Figure 10.5 is shown in Figure 10.6. As you can see, the program initializes the Error Code Bytes Available field, QUSBPRV, to zero, and then calls the Send Program Message API. On the call, we specify that we want message MSG0001, from the message file MSGS in SOMELIB (QualMsgF), sent as a *STATUS message to the external (*EXT) message queue associated with the job.

```
h dftactgrp(*no)

DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++
d/copy qsysinc/qrpglesrc,qusec

dSndPgmMsg        pr                  extpgm('QMHSNDPM')
d MsgID                          7    const
d QualMsgF                      20    const
d MsgDta                     65535    const options(*varsize)
d LenMsgDta                     10i 0 const
d MsgType                       10    const
d CallStackEntry             65535    const options(*varsize)
d CallStackCntr                 10i 0 const
d MsgKey                         4
d QUSEC                               likeds(QUSEC)
d LenCSE                        10i 0 const options(*nopass)
d CSEQual                       20    const options(*nopass)
d DSPWaitTime                   10i 0 const options(*nopass)
d CSEType                       10    const options(*nopass)
d CCSID                         10i 0 const options(*nopass)

dSleep            pr            10u 0 extproc('sleep')
d Seconds                       10u 0 value
```

Figure 10.6: Display a status message (part 1 of 2).

```
dQualMsgF         ds
d MsgF                           10    inz('MSGS')
d MsgL                           10    inz('SOMELIB')

dMsgKey           s               4

CL0N01Factor1+++++++Opcode&ExtFactor2+++++++Result++++++++Len++D+HiLoEq
 /free
  QUSBPRV = 0;
  SndPgmMsg( 'MSG0001' :QualMsgF :' ' :0 :'*STATUS' :'*EXT' :0
             :MsgKey :QUSEC);
  Sleep(10);
  *inlr = *on;
  return;
 /end-free
```

Figure 10.6: Display a status message (part 2 of 2).

If that was all the program did, you would most likely never see the message. It would be shown and removed so quickly that you would have a difficult time even knowing it was there. To make sure you can see the message, we use the sleep API to cause the program to delay 10 seconds before ending. In a production environment, the program would presumably be off doing database I/O, heavy computations, or some such. But as we didn't have anything like that to do in FIG10_6, we elected just to let the program sleep.

The sleep API documentation can be found in the Information Center in the System API category "Unix-Type APIs," and then subcategory "Signal APIs." The parameter description for sleep is shown in Figure 10.7.

```
unsigned int sleep (unsigned int seconds)
```

*Figure 10.7: The **sleep** API.*

The sleep API

The sleep API suspends a job, or more technically correct a thread, for a specified number of seconds. The job consumes no CPU while sleeping; it's just delayed for the number of seconds you specify. There are related APIs that can cause the requested sleep period to be shorter than specified. These are also described in the documentation for the "Signal APIs" subcategory. In FIG10_6, we expect to sleep for the full specified time period. You could, though, write an application where you want to sleep for *X* seconds, but be awakened if work becomes available for you to process before *X* seconds have passed.

The sleep API accepts one parameter, Seconds, which is defined as an unsigned integer (10u 0). The Seconds parameter specifies the number of seconds the job should be suspended. The sleep API defines a return value that is also an unsigned integer value. If the return value is zero, sleep suspended the job for the full requested period of time. If the return value is a positive value, it represents the number of seconds remaining of the requested period of time. That is, if you asked for 10 seconds but were awakened after seven seconds, the return value would be three. If the return value is negative one, an error was encountered, and ERRNO should be examined to gather additional information. (Access to ERRNO is described in chapter 14.) In program FIG10_6, we simply ignore the return value of the sleep API.

Sending Program Messages with Replacement Data

To add a bit more to FIG10_6, and to make more realistic use of a status message, let's now send a message with replacement data. The replacement data will be a variable indicating to the user an estimated time to completion. For the sake of good form, we'll also send a completion messages when the program is done. The new messages are shown in Figure 10.8.

```
ADDMSGD MSGID(MSG0002) MSGF(SOMELIB/MSGS) MSG('Job is doing some work.
Remaining time is &1 seconds.') FMT((*BIN 4))

ADDMSGD MSGID(MSG0003) MSGF(SOMELIB/MSGS) MSG('Job has completed.')
```

Figure 10.8: Create two more messages.

Message MSG0002 is defined as having one replacement variable (&1). The variable &1 is further defined as being a 4-byte binary value. Figure 10.9 shows the modified FIG10_6, which sends status messages reflecting the current state of the program and a completion message when finished.

```
h dftactgrp(*no)

DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++
d/copy qsysinc/qrpglesrc,qusec

dSndPgmMsg        pr                  extpgm('QMHSNDPM')
d MsgID                           7    const
```

Figure 10.9: Display a status message with replacement data along with a completion message (part 1 of 2).

```
d QualMsgF                       20    const
d MsgDta                      65535    const options(*varsize)
d LenMsgDta                      10i 0 const
d MsgType                        10    const
d CallStackEntry              65535    const options(*varsize)
d CallStackCntr                  10i 0 const
d MsgKey                          4
d QUSEC                                likeds(QUSEC)
d LenCSE                         10i 0 const options(*nopass)
d CSEQual                        20    const options(*nopass)
d DSPWaitTime                    10i 0 const options(*nopass)
d CSEType                        10    const options(*nopass)
d CCSID                          10i 0 const options(*nopass)

dSleep            pr             10u 0 extproc('sleep')
d Seconds                        10u 0 value

dQualMsgF         ds
d MsgF                           10    inz('MSGS')
d MsgL                           10    inz('SOMELIB')

dMsgTxt           ds
d Seconds                        10i 0

dMsgKey           s               4

CL0N01Factor1+++++++Opcode&ExtFactor2+++++++Result++++++++Len++D+HiLoEq
 /free
  QUSBPRV = 0;
  Seconds = 10;
  SndPgmMsg( 'MSG0002' :QualMsgF :MsgTxt :%size(MsgTxt)
             :'*STATUS' :'*EXT' :0 :MsgKey :QUSEC);
  Sleep(5);
  Seconds = 5;
  SndPgmMsg( 'MSG0002' :QualMsgF :MsgTxt :%size(MsgTxt)
             :'*STATUS' :'*EXT' :0 :MsgKey :QUSEC);
  Sleep(5);
  SndPgmMsg( 'MSG0003' :QualMsgF :' ' :0 :'*COMP' :'*EXT' :0
             :MsgKey :QUSEC);
  *inlr = *on;
  return;
 /end-free
```

Figure 10.9: Display a status message with replacement data along with a completion message (part 2 of 2).

FIG10_9 defines the data structure MsgTxt to hold the message replacement data for message MSG0002. This data structure has one sub-element, Seconds, which is defined as a 4-byte integer (10i 0). Seconds reflects the amount of time remaining for FIG10_9 to continue running. MsgTxt is then passed as the third parameter, along with the size of MsgTxt as the fourth parameter, on the first two calls to the Send Program Message API. The first two calls use message MSG0002 as a status message indicating the amount of time remaining until the task finishes. At the end of the program, FIG10_9 sends the completion message MSG0003.

Using Retrieve Message to Read Message Descriptions

Many applications have hundreds, if not thousands, of message descriptions stored in message files. The next sample program will show you how to read these message descriptions and process them within your application program. Specifically, the sample program will scan all messages in a specified message file, looking to see if a given word is found in either the first level or second level text of the message. If found, the program will print the message ID. Many more types of applications are possible with this technique—merging message descriptions across message files, printing all messages within a given range, etc.—but the prerequisite for any of these types of applications is the ability to read a message file. You will see that by using the Retrieve Message API (QMHRTVM), this is fairly straightforward.

Table 10.4 shows the parameters for the Retrieve Message API. Although it has quite a few parameters, the API itself is just like any other retrieve API.

The first parameter is the receiver variable where the Retrieve Message API returns data to the API caller. The second parameter is the size of the receiver variable, and the third parameter is the format you want used in returning the data.

The Retrieve Message API supports four formats. Formats RTVM0100 and RTVM0200 are primarily used when you are reading a specific message description and you know in advance the ID of the message you want. Formats RTVM0300 and RTVM0400 can be used for either keyed access by the message ID (similar to how RTVM0100 and RTVM0200 are used) or when sequentially reading through the message file. We will be sequentially reading all messages in the message file, so we will use the RTVM0300 format. This format returns all of the information necessary to the application. Format RTVM0400 also returns all of the necessary information, but it also returns additional information that we

Table 10.4: The Send Program Message API (QMHSNDPM)			
Parameter	**Description**	**Type**	**Size**
1	Message Information	Output	Char(*)
2	Length of Message Information	Input	Binary(4)
3	Format Name	Input	Char(8)
4	Message ID	Input	Char(7)
5	Qualified Message File Name	Input	Char(20)
6	Replacement Data	Input	Char(*)
7	Length of Replacement Data	Input	Binary(4)
8	Replace Substitution Values	Input	Char(10)
9	Return Format Control Characters	Input	Char(10)
10	Error Code	I/O	Char(*)
Optional Parameter Group 1			
11	Retrieve Option	Input	Char(10)
12	CCSID to Convert to	Input	Binary(4)
13	CCSID of Replacement Data	Input	Binary(4)

don't need. To get the best performance, select the format that provides the least amount of unnecessary information. Figure 10.10 shows the QSYSINC definition for format RTVM0300.

The fourth parameter is the message identifier for the message description we want returned. The fifth parameter is the qualified message file name.

The sixth parameter is the replacement data for the message. When retrieving the message description, you can optionally provide the replacement data for the substitution variables defined in the message. The values passed in this parameter would be used in the same way that the third parameter of the Send Program Message API was used when sending status messages in FIG10_9, and the values will be merged into the message text returned in the receiver variable. The seventh parameter is the length of the replacement data in the sixth parameter.

The eighth parameter allows you to control whether or not the substitution variables defined in the message are replaced by the replacement data provided in the sixth parameter. A *NO indicates that you want the substitution variables (the literals &1, &2, &3, etc.) returned. A *YES indicates that you want the substitution variables replaced by the replacement data provided in the sixth parameter.

The ninth parameter allows you to specify whether or not format-control characters (such as &N and &P) are returned in the message second level text. A *NO indicates you do not want the format control characters returned, a *YES indicates you do.

The tenth parameter is the standard error-code structure.

The eleventh parameter, Retrieve Option, is what allows us to sequentially read through a message file. Three special values can be used. The first, *MSGID, indicates that we want the API to return the message description associated with the message ID passed in the fourth parameter. This value is not going to help us initially in our current task, as we don't know the message IDs. The second special value, *NEXT, indicates that we want the API to return the message description associated with the next message following the message ID passed in the fourth parameter. This special value is closer to what we're looking for. All we need now is to be able to identify the first message ID within the message file, and then we can use *NEXT to read through the rest of the file. The third special value, *FIRST, retrieves the first message in the message file and ignores the fourth parameter. With this, we now have the necessary retrieval options to read all message descriptions stored in any message file!

The remaining two parameters are related to CCSID conversions of the message text and CCSID processing of the replacement data. These are important functions, but they're not something we need to be concerned with for our sample program.

```
DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++
D*****************************************************************
D*Type Definition for the RTVM0300 format.
D****                                                          ***
D*NOTE: The following type definition only defines the fixed
D*   portion of the format.  Any varying length field will
D*   have to be defined by the user.
D*****************************************************************
```

Figure 10.10: Format RTVM0300 from QSYSINC/QRPGLESRC member QMHRTVM (part 1 of 3).

```
DQMHM030000       DS
D*                                                Qmh Rtvm RTVM0300
D QMHBR03                 1      4B 0
D*                                                Bytes Return
D QMHBAVL09               5      8B 0
D*                                                Bytes Available
D QMHMS09                 9     12B 0
D*                                                Message Severity
D QMHAI00                13     16B 0
D*                                                Alert Index
D QMHAO03                17     25
D*                                                Alert Option
D QMHLI02                26     26
D*                                                Log Indicator
D QMHMID                 27     33
D*                                                Message ID
D QMHERVED19             34     36
D*                                                Reserved
D QMHNRDF                37     40B 0
D*                                                Number Replace Data
D QMHSIDCS07             41     44B 0
D*                                                Text CCSID Convert
D QMHSIDCS08             45     48B 0
D*                                                Data CCSID Convert
D QMHCSIDR07             49     52B 0
D*                                                Text CCSID Returned
D QMHORT                 53     56B 0
D*                                                Offset Reply Text
D QMHLRRTN00             57     60B 0
D*                                                Length Reply Return
D QMHLRAVL00             61     64B 0
D*                                                Length Reply Availa
D QMHOMRTN               65     68B 0
D*                                                Offset Message Retu
D QMHLMRTN04             69     72B 0
D*                                                Length Message Retu
D QMHLMAVL04             73     76B 0
D*                                                Length Message Avai
D QMHOHRTN               77     80B 0
D*                                                Offset Help Returne
D QMHLHRTN04             81     84B 0
D*                                                Length Help Returne
D QMHLHAVL04             85     88B 0
D*                                                Length Help Availab
D QMHOF                  89     92B 0
D*                                                Offset Formats
```

Figure 10.10: Format **RTVM0300** *from* **QSYSINC/QRPGLESRC** *member* **QMHRTVM** *(part 2 of 3).*

```
D QMHLFRTN                93     96B 0
D*                                     Length Formats Retu
D QMHLFAVL                97    100B 0
D*                                     Length Formats Avai
D QMHLFE                 101    104B 0
D*                                     Length Format Eleme
D*QMHRSV203              105    105
D*
D*                                 Varying length
D*QMHDR00                106    106
D*
D*                                 Varying length
D*QMHSSAGE04             107    107
D*
D*                                 Varying length
D*QMHMH04                108    108
D*
D*                                 Varying length
D*QMHRDF                  18     DIM(00001)
D* QMHLSRD00               9B 0 OVERLAY(QMHRDF:00001)
D* QMHFSODP00              9B 0 OVERLAY(QMHRDF:00005)
D* QMHSVT00                10     OVERLAY(QMHRDF:00009)
D*
D*
D*                                 Varying length
```

Figure 10.10: Format RTVM0300 from QSYSINC/QRPGLESRC member QMHRTVM (part 3 of 3).

Format RTVM0300 returns quite a bit of information related to a message description. We're specifically interested in the following fields:

- The message ID, to identify the message containing the searched-for word
- The first-level message text, so it can be scanned for the searched-for word
- The second-level text, so it can be scanned for the searched-for word

In scanning the fields of the QSYSINC-provided QMHM030000 data structure, we quickly find that the message ID is returned in field QMHMID. However, the first- and second-level text fields for a message do not appear to be returned at a fixed location or with a fixed length. Given the variable-length nature of these fields, this makes sense. Instead, we find that the API returns an offset to the first level text, QMHOMRTN, the length of the first level text, QMHLMRTN04, an offset to the second level text, QMHOHRTN, and the length of the second level text, QMHLHRTN04. With this, we have everything we need.

Figure 10.11 shows how this information is used to scan for a given word within the first- and second-level text. Program FIG10_11 accepts one parameter, Word. This parameter is the word that the program will scan for in the retrieved message's first- and second-level text.

```
h dftactgrp(*no)

fqsysprt   o    f  132         printer

DName++++++++++++ETDsFrom+++To/L+++IDc.Keywords++++++++++++++++++++++
dFig10_11         pr
d Word                           10    const

dFig10_11         pi
d Word                           10    const

d/copy qsysinc/qrpglesrc,qmhrtvm
d/copy qsysinc/qrpglesrc,qlg
d/copy qsysinc/qrpglesrc,qusec

dGetMsg           pr                   extpgm('QMHRTVM')
d Receiver                        1    options(*varsize)
d LenReceiver                    10i 0 const
d Format                          8    const
d MsgID                           7    const
d QualMsgF                       20    const
d RplDta                      65535    const options(*varsize)
d LenRplDta                      10i 0 const
d RplSubs                        10    const
d RtnCtls                        10    const
d QUSEC                               likeds(QUSEC)
d RtvOption                      10    const options(*nopass)
d CCSIDTo                        10i 0 const options(*nopass)
d CCSIDRplDta                    10i 0 const options(*nopass)

dConvertCase      pr                   extproc('QlgConvertCase')
d Request                     65535    const options(*varsize)
d InputData                   65535    const options(*varsize)
d OutputData                      1    options(*varsize)
d InputDataLen                   10i 0 const
d QUSEC                               likeds(QUSEC)

dReceiver         ds                   qualified
d Base                                 likeds(QMHM030000)
d Variable                    10000
```

Figure 10.11: Find all messages using a particular word, and print the message ID (part 1 of 3).

```
dMsgTxt           s            132    based(MsgTxtPtr)
dHlpTxt           s           3000    based(HlpTxtPtr)
dNewMsgTxt        s            132
dNewHlpTxt        s           3000
dQualMsgF         s             20    inz('QCPFMSG   QSYS       ')
dNewWord          s             10
dMsgID            s              7
dHit              s             10u 0
dwait             s              1

CL0N01Factor1+++++++Opcode&ExtFactor2+++++++Result++++++++Len++D+HiLoEq
 /free
  if %parms = 0;
     dsply 'Required word parameter not passed' ' ' wait;
  else;
  QUSBPRV = 0;

  // Convert the parameter value to uppercase
  QLGIDRCB00 = *loval;           // set input structure to x'00'
  QLGTOR02 = 1;                  // use CCSID for monocasing
  QLGIDOID00 = 0;                // use the job CCSID
  QLGCR00 = 0;                   // convert to uppercase
  ConvertCase( QLGIDRCB00 :Word :NewWord
               :%len(%trimr(Word)) :QUSEC);

  // Get the first message in QualMsgF
  GetMsg( Receiver :%size(Receiver) :'RTVM0300' :' ' :QualMsgF
          :' ' :0 :'*NO' :'*NO' :QUSEC :'*FIRST' :0 :0);

  dow (Receiver.Base.QMHMID <> *blanks);

       // Convert to uppercase the first level message text
       if (Receiver.Base.QMHLMRTN04 > 0);
          MsgTxtPtr = %addr(Receiver) + Receiver.Base.QMHOMRTN;
          ConvertCase( QLGIDRCB00 :MsgTxt :NewMsgTxt
                       :Receiver.Base.QMHLMRTN04 :QUSEC);
          Hit = %scan(%trim(NewWord) :NewMsgTxt);
          NewMsgTxt = *blanks;
       endif;

       // if not found in first level text, look at second level text
       if (Hit = 0) and Receiver.Base.QMHLHRTN04 > 0;
          HlpTxtPtr = %addr(Receiver) + Receiver.Base.QMHOHRTN;
          ConvertCase( QLGIDRCB00 :HlpTxt :NewHlpTxt
                       :Receiver.Base.QMHLHRTN04 :QUSEC);
          Hit = %scan(%trim(NewWord) :NewHlpTxt);
          NewHlpTxt = *blanks;
       endif;
```

Figure 10.11: Find all messages using a particular word, and print the message ID (part 2 of 3).

```
      if (Hit <> 0);
         MsgID = Receiver.Base.QMHMID;
         except GotOne;
         Hit = 0;
      endif;

      // Get the next message
      GetMsg( Receiver :%size(Receiver) :'RTVM0300'
             :Receiver.Base.QMHMID :QualMsgF :' ' :0 :'*NO' :'*NO'
             :QUSEC :'*NEXT' :0 :0);
  enddo;
  endif;

  *inlr = *on;
  return;
 /end-free

oqsysprt   e            GotOne
o                       MsgID              10
```

Figure 10.11: Find all messages using a particular word, and print the message ID (part 3 of 3).

Program FIG10_11 first tests to see if the Word parameter was passed. If not, it DSPLYs appropriate error text. If a parameter was passed, the program initializes the Bytes Provided field of the error-code structure to zero to indicate that errors should be returned as exception messages to the program. The program then uses the Convert Case API, QlgConvertCase, to convert the variable Word to uppercase and store the results in variable NewWord. (The Convert Case API was discussed in chapter 9.)

The program then calls the Retrieve Message API, QMHRTVM, requesting that the *FIRST message in message file QualMsgF (QCPFMSG in QSYS) be returned in the receiver variable, using format RTVM0300. The receiver variable is defined as being LIKEDS(QMHM030000), which is the QSYSINC include for format RTVM0300, followed by 10,000 bytes for the variable data that might be returned by the API. This variable data will include the first-level text, the second-level text, the substitution variable formats, and more. Based on the current maximums for these fields, we feel that 10,000 bytes would be sufficient. (If you really wanted to be sure you had sufficient storage, you could enhance this program by using the dynamic receiver variable approach introduced in chapter 2.)

After the message has been retrieved, the program falls into a do-while (DOW) loop that is exited when the returned messaged ID, Receiver.Base.QMHMID, is all blanks. When

using the special values *FIRST or *NEXT for the Retrieve Options parameter, a blank message ID field indicates that the end of the message file has been encountered.

If a message description was returned, and first-level text exists for the message (Receiver.Base.QMHLMRTN04 > 0), then the first-level text is accessed by setting the pointer MsgTxtPtr to the address of the Receiver variable (%addr(Receiver)), plus the offset to the message text value (QMHOMRTN). As MsgTxtPtr is defined as the BASED pointer for the MsgTxt field, the program now has direct access to the first-level text of the message. The first-level text, MsgTxt, is now converted to uppercase by the QlgConvertCase API. The uppercased results are stored in the work field NewMsgTxt.

FIG10_11 then scans the uppercased NewMsgTxt field for occurrences of the uppercased value stored in the NewWord field. If that value is found in NewMsgTxt, the field Hit is set to a positive value. Technically, this positive value is the first position in NewMsgTxt where the value was found, but we really don't care about the position (in this application, anyway). All we are interested in is whether the value is found or not. If Hit is zero, the value was not found in NewMsgTxt. The program sets NewMsgTxt to blanks after the %scan to make sure there are no trailing characters remaining in NewMsgTxt when a subsequent call to the Retrieve Message API has a message with shorter first-level text than the current message. If the value is not found in NewMsgTxt (Hit = 0), the same process is applied to the second-level text. If the word *is* found in either the first- or second-level text (Hit <> 0), the message ID is written to the QSYSPRT printer file.

Whether the word is found or not, the program next calls the Retrieve Message API asking for the *NEXT message description following the current message ID (Receiver.Base.QMHMID is being passed as the fourth parameter of the API call). It then continues the DOW loop. When all messages have been processed, the program ends.

Summary

This chapter provides some tools and examples for using message-handling APIs to watch for a message, send a message, and read message descriptions from a message file. However, there are many more APIs that deal with handling messages. This chapter barely scratches the surface of all that is available. Be sure to explore the message APIs on your own.

Check Your Knowledge

Earlier in this chapter, you saw how to send status and completion messages. Add two new messages to the SOMELIB/MSGS message file. The first message is message description MSG0006, with a first-level text of "I found a problem with my input." The second message is MSG0007 with a first-level text of "This problem has caused me to stop running."

Write a program that sends message MSG0006 as a diagnostic message to your program's caller. This diagnostic message should be followed by message MSG0007 as an escape message to your caller. To get you started, your program's caller is one call stack entry above your program boundary in the call stack.

One possible solution to this task can be found in Figure D.10.B of appendix D.