ANGLO-AUSTRALIAN OBSERVATORY AAO/IMP_MANUAL_8
DRAMA Software Report 8
Version 1.4.4

Keith Shortridge
9-Mar-2000

Interprocess Message Passing (IMP) System

Contents

1 Introduction

This document introduces and describes the IMP message-passing and task-loading system. IMP is intended to provide a communications system for real-time instrumentation tasks operating over a network. IMP is now at version 1.4.2 and is reasonably stable. However, it is a system that is still being tested and whose design may still change slightly as more people use it, since each new application tends to show up ways in which the system can be improved

1.1 Do I have to read all this?

This is a rather long document, and I'm all too aware that is isn't easy reading. It has grown as IMP has grown, and IMP is now rather a large system. The best thing is probably to read the 'Introduction', 'Philosophy' and 'General overview' sections and then to start looking at some of the example code. The simplest programs of all are the two test programs supplied with IMP, `test1.c` and `test2.c`. After that, a look at `ttest.c` and `atest.c` will get you pretty deep into the way straightforward IMP programs look. Of course, just to look at the source you will need to get hold of the IMP code, so you need to look at the start of the section on 'Installing IMP' to find out how to do that. After that, you can work through the rest of this document as you like. It wasn't really written to be read in sequence.

1.2 What does IMP do?

IMP provides a basis on which real-time instrumentation systems can be built. Nowadays, such systems are built as a set of modules (tasks, processes, or whatever terminology you prefer), often running on more than one computer. In many cases, the various computers involved may be quite different — from different manufacturers, even running different operating systems. This is because different systems have different strengths. UNIX is a good system to use for user interfaces, because of the variety of development tools it provides for such things. Real-time kernels such as VxWorks provide the 'hard' real-time response needed to control actual hardware, and can be built into VME or similar systems systems with all the control hardware that implies. There may be a need to use cheap, PC-based, hardware running either Linux or a variant of Windows. There may be historical reasons why a particular system is being used; AAO, for example, has a large amount of instrumentation software that runs under VMS.

To cooperate, the tasks running on these systems need to communicate, either by sharing data in some sort of noticeboard, or by sending messages to one another; or both. Noticeboards are convenient, but they are difficult to implement on distributed systems. Systems based on the sending of messages from task to task lend themselves naturally to the sort of 'event-driven' design needed by most real-time systems. An event — often externally generated, such as an interrupt from a device — causes a message to be sent to a module that knows how to handle such an event. In turn, other messages can be sent, triggering other actions that may eventually cause more external events. An external event may be the movement of a mouse on a screen, the click of a mouse button, or it may be the closing of a relay or triggering of a microswitch on a piece of equipment. It may be a timer triggering after a pre-determined time.

IMP provides a set of routines that allow messages to be sent between tasks. Most systems provide system routines that do this; UNIX, for example, has named pipes, message queues, and the like. What IMP provides is a set of routines that work between tasks on disparate, distributed, systems. IMP allows an interrupt service routine on a VxWorks system to send a message to a user interface task on a machine running UNIX, without either needing to know anything about the remote task other than the name under which it registered with the IMP system. Moreover, IMP is written to use the fastest available method for sending a message from one task to another on the same machine, so it has definite performance advantages over schemes that use a commonly available but sometimes inefficient set of routines such as sockets for all communication. It also always operates in a 'non-blocking' mode, never requiring a task to wait for a message to be received before control is passed back to the program.

These are all characteristics that experience has shown are needed for real-time systems.

IMP also includes comprehensive facilities for the running of tasks on both local and remote machines. This is not something that will be needed by programs that have already been loaded in some other way and merely want to use IMP for inter-process communication. However, much more robust multi-process systems can be built if the task loading is tightly coupled to the inter- process communications, since this makes it easier for problems associated with task loading (crashing of tasks, load failures, etc) to be reported to the requesting tasks.

1.3 IMP goals

Originally, the design of IMP was intended to meet the following criteria:

Speed The IMP routines are supposed to be fast; anything in a real-time system usually needs to be fast, and you should not be slowed down by using IMP. For example, in a VxWorks interrupt routine, you should feel quite happy to send a message using ImpSend(), knowing that it is not significantly slower than using the standard VxWorks routines such as msgQSend().

Multiple machines IMP should be able to run on any machine we need it to run on; at present the list is: VMS on Vaxes, OpenVMS on DEC Alphas, SparcStations running SunOS or Solaris, DEC Alphas running OSF/1 or OpenVMS, HP-UX 10, Linux, VxWorks. A version for DecStations running Ultrix exists but is no longer being actively supported (because AAO no longer has such machines — it could be revived if needed). A version for the Win32 API (Windows 95/98/NT) exists although it is still slightly experimental. It should be possible to port it to any sufficiently capable operating system (but not necessarily to single-tasking systems such as MS-DOS or the MacOS — although the new MacOS X should prove an easy target, when it becomes available).

Networking The system should be inherently able to work in a distributed system – a call to ImpSend() should look the same whether the target task is on the same machine or in a different continent. (Although the response times may be slower in the latter case.)

Message size There should be no artificial limitation on the size of the messages that can be sent using IMP.

Non-blocked operation IMP routines should not block — they should return immediately to the caller, and not have to wait for some event external to the process. A task should never be unable to respond to outside events as a result of a call to an IMP routine. (Although since getting an IMP message is seen as the main, indeed often the only, way a task is notified of outside events, a routine such as ImpRead() normally would block.)

Waiting without overheads A task waiting for a new message to be sent to it should be dormant, using minimal system resources; this rules out polling for new messages.

Reliability The IMP system should not crash, and it should also be able to survive when tasks using it crash for reasons of their own. Messages should be delivered reliably, and if a message cannot be delivered, the sending task should be notified of the fact.

Interrupt level working Any message that needs to be sent should in principle be able to be sent via IMP routines. So it should be possible to call them from interrupt level code on VxWorks, from an AST routine under VMS, or from a UNIX signal handler.

How well these grand aims have been achieved is discussed at the end of this document, in the section on the current status of the system.

1.4 How has IMP been used?

A little background may be interesting, if not entirely relevant.

IMP was designed as one of the sub-systems of the 'DRAMA' system written at the Anglo-Australian Observatory (AAO) to control a new instrument known as the 'two degree field' (2dF). Details can be found at the AAO web site (`www.aao.gov.au`), but briefly this involves a large optical corrector at the prime focus of the 3.9 metre Anglo-Australian Telescope (AAT). This provides a field of view covering two degrees on the sky — a far wider field than available on most large telescopes. Such a field will typically contain some thousands of galaxies, and one of the aims of 2dF is to measure the redshifts of as many galaxies as possible in order to do cosmological surveys, probing the large-scale structure of the universe. To do this, optical fibres are placed in the field at the predetermined positions of the images of up to 400 galaxies (limited by the number of fibres in the system). These feed the light from the galaxies into a pair of spectrographs and from the spectra of the galaxies their redshifts can be determined. The positioning of the fibres is done by a 'pick and place' robotic positioner controlled by a VME system running VxWorks, as are the mechanisms in the spectrographs. The user interfaces and on-line data reduction system run on UNIX workstations, and the detector control software runs (for historical reasons) on a VMS VAX.

The main point of this is to indicate that there was a need for an instrumentation environment that could handle the real-time requirements of a VME robotic system, that could also transmit large images from CCDs or from guiding cameras, that could link together tasks running on VxWorks, UNIX and VMS. The system that emerged, 'DRAMA' could do all these things, and at its core are mechanisms that allow tasks to send arbitrarily complex structured messages between tasks on a variety of machines. The messages may contain multi-megabyte images, or single status words, or complex hierarchical information about instrument settings. These messages are encoded using a DRAMA subsystem called '`SDS`' (Structured Data System) and are transmitted using IMP.

So IMP was originally designed to be a part of a much larger system, DRAMA. However, it is a self-contained part of DRAMA, and organisations outside AAO have picked it up and found it useful within their own systems. It is relatively efficient, and is not restricted to UNIX; working on a system as different as VxWorks is clearly an attraction.

2 Philosophy

In a DRAMA system, which is where IMP originated, most tasks do not make direct calls to IMP at all. The operation of IMP is hidden by a higher layer of code, called DITS (Distributed Instrument Tasking System). DITS sees the various tasks on a system as boxes that can support multiple named 'actions', which have parameters. These 'actions' are invoked by calls to DITS routines, and can interract (an action can be cancelled, for example, or can report its progress to other tasks) through calls to other DITS routines. DITS generates structured messages using SDS and sends them to other tasks using IMP. The main reason most applications do not call IMP directly is that they do not know (or need to know) the conventions used by DITS to format these messages. After all, the systems in question may have different byte orders, different floating point formats, different alignment requirements, different word lengths. Just being able to send a message to a remote machine is one thing; you need to be sure it will be understood when it arrives. IMP handles the first, but a system such as SDS is needed for the second.

This means that when IMP was designed, it was felt that it was important to provide comprehensive information about what was going on in the system. This was more important than making the interface elegant or simple, since it was not generally application level code that was going to be calling IMP. The result is that IMP is a system that always reminds me of the Pompidou Centre in Paris — a building that has all its plumbing on the outside. IMP doesn't hide much from the code that uses it; IMP has a lot of internal 'system' messages which are connected with its internal handshaking, but these are all visible to the user. When you read a message with IMP, it may be one your code was expecting, or it may be one of these IMP system message. You can tell a system message by its negative type code, and usually all you do is call an IMP utility routine to process it (this is `ImpSystemMessage()`). This utility routine decodes the message and tells you something about it. In some cases you actually have to do something yourself, in others all you do is ignore the message. This isn't all that different to what happens in some windowing systems; all of these have a central point where all events are handled and doled out to their various service routines, but some (like IMP) require that central point to be in application code and others have it hidden in the system layer. Having it in application code may be inelegant, but it does provide more control (application code being debugged, for example, can log all system messages, because it sees all system messages).

The IMP routines have been designed around a model of a task driven by the arrival of incoming messages, and no other external events. (Note, however, that it is possible to run IMP so that a task can include IMP events in an X-Windows event loop, or — on some systems — to arrange to wait for a combination of IMP and other events, using, for example, a call to a system routine such as '`select()`') This model envisages a task as constantly returning to its message queue(s), waiting for a new message, processing that message quickly and returning to wait for another message. The only place where the task blocks is when waiting for a new message to arrive. This means that a task that is not being sent messages can be completely quiescent, using minimal system resources (polling for new messages is not needed, for example), but is always able to respond to a new message sent to it. IMP can, of course, be used in tasks based around quite different models, but this is the one for which it was originally designed.

3 General Overview

This section is intended to provide an overview of the IMP system, giving a very brief but comprehensive idea of the sort of facilities it provides. All the user-callable routines in IMP are mentioned at least once in this section.

Tasks register themselves with the IMP system using an identifying name. A task running on one machine can locate a task registered on another machine (it issues a 'locate' request for the task, specifying the machine and its registered name). Once a remote task is located in this way, it is registered on the local machine under the same name. Once a task is registered on a machine, either directly or as the result of a locate request, other tasks can communicate with it, the IMP system handling communication with tasks on remote machines in just the same way as for local tasks. Initial registration of a task happens when the task calls `ImpRegister()`, and location of a remote task is accomplished through a call to `ImpNetLocate()`. A task may specify a small amount of additional information about itself using `ImpSetDetails()` and other tasks may access this using `ImpGetDetails()`, but this is provided only as a convenience function for writers of collections of cooperating tasks — this additional information is not used by the IMP system itself.

Communication between tasks is 'connected', a task establishing a connection with another task that it specifies by name. The task may be either a local task, or a remote one previously registered as a result of a call to `ImpNetLocate()`. Once a task establishes a connection with another task, it receives a connection number which it uses to send messages to that task. A connection is established through a call to `ImpConnect()`.

A connection may be one-way or two-way. A one-way connection gives a task a path to another task, but does nothing connected with allowing the other task to reply. The other task can tell which task has sent it a message, and can choose to establish a second one-way connection that it can use to reply to the sending task, but this is up to it. In a real-time system, for example, an interrupt service routine, for example, may service an interrupt and then send a message to another task to indicate that the interrupt has occurred. Perhaps on the interrupt, the service routine reads in an amount of data and tells a display task where to find it. Such a routine will almost certainly not want to have to wait for a reply from the display task, and a one-way connection, which has minimal overheads, is what it would want to use.

A two-way connection is really just two one-way connections, but they are established at the same time and this overcomes a number of handshaking problems. For example, if one task connects to another and starts to send it messages, then for the other task to reply to them it must first establish a connection and it will have to wait for that to be established. While it is waiting for the connection it may have to stack the messages it is receiving and to which it wants to reply. This is overcome by having both connections made at the same time. A two-way connection has to be accepted by the second task through a call to `ImpAcceptConnect()` or `ImpAccept()`. A task can call `ImpConnection()` to see if it has an existing connection to a specified task.

`ImpConnectInfo()` can be used to find out information about a connection, such as which task is at the other end and how much space has been allocated for messages and how much remains unused. `ImpMessages()` can interpret the 'number of bytes available' values returned by `ImpConnectionInfo()` in terms of the number of messages of a specified size this can accomodate (it allows for overheads such as message headers). A connection can be closed through a call to `ImpCloseConnect()`.

Once a connection is established, a task can send messages to that other task, specifying the connection number in question. A task can receive messages from many tasks at once, and may also receive messages generated by the IMP system itself. When a task issues a read command it may request an immediate return whether or not there are messages waiting, or may request that the read should block until a new message arrives (the latter would be more usual in the task model described above). In either case, the first unread message sent to the task is read. A task may also specify a time-out period, after which the read routine will return even if no messages have been received. A timeout is specified in the most efficient format for the system being used, and the routine `ImpDeltaTime()` should be used to generate timeout values for use by the reading routines.

Messages are sent using a call to either `ImpSend()`, or `ImpSendPtr()` and read using either `ImpRead()` or `ImpReadPtr()`. It is possible — although not usual — to use these routines to 'peek' ahead into the message queues and so access messages out of the normal sequence; this is discussed separately in the section on 'message peeking'. A message that has been 'peeked' at can be flagged as if it had been read normally through a call to `ImpSetAsRead()`.

`ImpSend()` is passed a pointer to a message buffer supplied by the caller, and it sends this message. This will involve a copy out of the caller's buffer; for a very long message (and messages can be megabytes long) this can be a noticeable overhead. The alternative `ImpSend-Ptr()` call is passed the length of the message to be sent and returns a pointer to a sufficiently large memory area already in the IMP system. The caller can then construct the actual message in that area and then call `ImpSendEnd()` to finally send the message. Similarly, once a message is received, the receiving process can call `ImpRead()` to read the message into a message buffer supplied by the caller, but this again involves a copy of the message. Alternatively, the receiving process can call `ImpReadPtr()` which will just return a pointer to the message. The receiving process can then handle that message *in situ* and call `ImpReadEnd()` to indicate when it has finished with the message.

If a task calls `ImpReadPtr()` to get a pointer to the next message waiting for it, it can start to process that message and then make another call to `ImpReadPtr()` before making the call to `ImpReadEnd()` that releases the first message. In this case, the connection that the first message was read from will be blocked, but a message waiting on another connection can be read. This facility is intended mainly for use by the IMP networking tasks. Any task that uses this must set a 'multiple concurrent read' flag in all its calls to `ImpReadPtr()`.

From time to time, messages generated by the IMP system itself may be received by a task. These have a negative 'type' field associated with them and should be processed by passing them to `ImpSystemMess age()`. This routine will decode the message, handle it so far as is possible automatically, and will fill out a structure that the calling routine can examine to determine if it needs to do anything as the result of the message. System messages are used to report errors in the sending of messages, unexpected disconnections or crashes of other tasks, etc.

Although a task should only rarely need to know if another task is on the same machine or not, IMP provides the routine `ImpIsTaskLocal()`. This can be passed an IMP task identifier (as returned, for example by `ImpSystemMessage()`) and will return true or false depending on whether or not the task in question is on the local machine.

A task can find how many messages are waiting for it by calling `ImpMessageCount()`. The routine `ImpFromTask()` identifies the task that sent a newly-read message.

A task can queue a message to be sent to itself after a specified delay by calling `ImpQueueRe-minder()`. Although `ImpRead()` and `ImpReadPtr()` can take a time-out value to prevent a task hanging indefinitely while waiting for a message, setting up such a time-out can have noticeable overheads and it may often be better for a task expecting to receive a large number of messages in a short time to simply queue regular reminders to itself and check after each one whether or not more message have arrived since the last timeout. Messages queued in this way can be cancelled before they are sent, if necessary, by calling `ImpClearReminder()`

A connection may be 'flow controlled'. When such a connection is established, the IMP system itself provides additional handshaking that can be used to monitor the rate at which the target task is reading the messages sent on the connection. The sending task can call `ImpCheckSend()` prior to any call to `Imp Send()` or `ImpSendPtr()` to make sure that there is enough space in the buffers used by the connection for the message it is about to send. If the IMP system cannot guarantee sufficient space for the message (on both the local and the remote machines) then `ImpCheckSend()` will flag this. The calling program can then call `ImpRequestNotify()` to request that it be notified (through a system message) when the connection is clear. The use of flow controlled connections is discussed in more detail in a later section.

`ImpRunTask()` allows a task to request that a new task be run on either the local machine or on a remote machine. A task that makes such a request will be notified of the progress of the loaded task through a series of system messages.

It is possible for a task to register itself in such a way that it can receive both IMP messages and X-events, allowing it to run an X user-interface. The information that will have to be passed to the X system to allow this to happen can be obtained through a call to `ImpGetXInfo()`. It is also possible to use this information on some systems to wait for a combination of IMP and other events. For example, under UNIX, the information returned by `ImpGetXInfo()` can be used directly in a call to `select()`. Note that this is system-dependent; it is not so easy under VMS, although it can be done.

If an IMP routine returns with a non-zero status code, it is possible to call `ImpErrorText()` to get a string describing the error. This provides an easy way of generating a useful error message, rather than just printing an unhelpful error code value.

There are a number of routines connected with the efficient transfer of memory mapped data. These allow bulk data to be sent from an area of shared memory on one machine as directly as possible (with no unnecessary copying of the data) into an area of shared memory on a remote machine. They also provide a protocol for sharing such data between tasks on the same machine. The routine `ImpDefineShared()` should be called to describe an area of shared memory to the IMP system. `ImpSendBulk()` can then be called to initiate the transfer of the contents of this shared memory area to another task. The target task will receive an IMP system message describing this shared memory data. Depending on the details of the transfer, the target task should then call either `ImpReadBulk()` or `ImpReadBulk()` to access the data. Shared memory may be released through a call to `ImpReleaseShared()`.

A task may be forcibly deleted from the system through a call to `ImpDeleteTask()`.

As it shuts down a task should detach itself from the system. This is done through a call to `ImpDetach()`.

IMP also provides a few 'convenience' routines that provide system-independent access to system-dependent things such as time services. These are not connected with IMP's main function of message passing and task loading, but had to be written for IMP's internal use and so may make it a little easier to write system-independent code. (They turned out to be useful when writing the programs used to test IMP.) These are `ImpNodeName()`, `ImpSetTime()`, `ImpElapsedTime()`, `ImpTimeNow()` and `ImpTimeSince()`.

Although a task normally registers under one specific name (the one supplied in its call to `ImpRegister()`, IMP also supports the concept of a 'Translator' task. A 'Translator' task is one that registers with the `IMP_ TRANSLATOR` flag set in its call to `ImpRegister()`. Such a task can act on behalf of a number of other named tasks, receiving messages meant for them and replying on their behalf. This was introduced as a means of supporting communication with already existing tasks that used some other communications mechanism; a translator task can intercept messages for them, send them on using whatever communications system they respond to, and arrange for messages to be sent back when they respond. In this case, the other tasks do actually exist, they just aren't able to sit directly on the IMP message system and need a translator task to intercede for them. However, a task could, for reasons of its own, register as a translator and act as if it were a number of different tasks; in this case the named 'tasks' for which the translator task acts are purely conceptual and do not exist outside the translator task itself. The test program '`ttran`' listed later in this manual is an example of such a task.

A number of additional IMP routines are provided to support 'Translator' tasks, and will probably not be called other than from 'Translator' tasks. These are `ImpProxyRegister()`, `ImpProxyDetach()`, `ImpInputNumber()`, and `ImpHandled()`. The topic of 'Translator' tasks is dealt with in more detail in a later section.

4 The IMP 'Master' task

A simple system can be set up using IMP where all communication is between processes running on the same machine and loaded by some external means. Such programs can simply call `ImpRegister()` and then connect with each other and pass messages quite happily.

However, if processes need to communicate across a network, or want to request that other processes be run on their behalf, a more complex structure needs to be established. IMP network communications is handled by a pair of network tasks on each machine (a 'receiver' task and a 'transmitter' task), and task loading and system supervision is handled by an IMP 'Master' task.

Once started, the IMP 'Master' task loads the network tasks and monitors them. Normally, it loads these from the same directory that it itself was loaded from; if they are to be found elsewhere, then the VMS logical name or UNIX environment variable IMP_DIR may be set to indicate the actual directory to be used. Under both VMS and UNIX this must translate (in the context of the Master task) to a full directory specification; under UNIX a final '/' is optional.

It also loads any other tasks as requested (by tasks calling `ImpRunTask()`) and monitors them on behalf of the requesting tasks. This provides a significant degree of robustness — if a task loaded by the 'Master' task crashes, the 'Master' task will be notified. It can then handle any tidying up of the task — it can locate and release any resources the task may have had, can clear any semaphores the task had taken, etc — and it can notify the original requesting task (even if the requesting task is on a remote machine). If the network tasks crash for any reason (not that this tends to happen) the 'Master' task will automatically reload them.

4.1 Bypassing the Master Task

Normally, the IMP Master task will perform any program loading required by the system. Having the one master task able to monitor and tidy up after all the tasks in the system makes for a more robust system, and frees individual tasks of the need to deal with task crashes and the like. However, in a simple system, such as a data reduction system running on a single machine — as opposed to a tightly-coded real-time data acquisition system, the additional safeguards provided by the Master task may not be necessary and the overheads it imposes (actually having to make sure the Master task is running) can be a problem.

For these circumstances, IMP allows a task to set the `IMP_MAY_LOAD` flag when it calls `ImpReceiver()`. This indicates that the task can do its own task loading if the Master task is not running. There is nothing else such a task needs to do. It will find that a call to ImpRunTask() will no longer fail with a 'Master task not present' error code, and it will get the same system messages indicating successful registration of its sub-task and eventual exit or crashing of the sub-task. There are a couple of minor restrictions:

1. There will be some system-dependent restrictions on the task, imposed by the need for it — rather than the Master task — to monitor the loaded task. Under UNIX, it must not set up a SIGCHLD signal handler, for example. There are no such restrictions under VMS or VxWorks.

2. The status of the sub-task once the loading task exits is now system-dependent. Under UNIX and VxWorks the sub-task will continue to execute; under VMS the sub-task really is a sub-task of the loading task, in the very meaningful sense that it will be deleted when its loading task exits.

5 IMP System messages

Often, a simple program can anticipate the order in which messages will be sent to it, and one can imagine code that consists of a dialogue with another task that consists of a predetermined sequence of calls to `ImpSend()` and `ImpRead()`, where the program knows for each call to `ImpRead()` just what message it expects to receive. This can obviously be accommodated by the IMP design. However, a more natural way to code the type of task that IMP was really intended for is to code it as an 'event- driven' task, with a main loop that just reads the next message, sees what sort of message it is and then calls the code intended to handle that message. Such a program is much better equipped to handle the asynchronous demands placed on real-time tasks, and is much less likely to be troubled by unexpected messages. In the IMP context, a task may well find that the next message it reads is an IMP system message — perhaps that a task has been located, or a task is asking to be notified when the calling task is able to receive more messages. The IMP design does not hide these messages from the task. `ImpRead()`, for example, could have been written to handle many of these messages automatically, but has in fact been written to return them as if they were ordinary messages. Each message sent through the IMP system has a type associated with it,and messages such as these can be identified as IMP 'system messages' from their tag values.

A program using IMP is expected to check messages read by `ImpRead()` and to see what type they are tagged as being. It should then pass IMP system messages (which have a negative type value) on to the routine `ImpSystemMessage()`. This will handle the message and decode it and return information about it to the caller. An example may make this clearer, showing the stages a program passes through when it establishes a connection with a remote task:

1. The program attempts to locate a named task on a remote machine, using `ImpLocate()` and specifying a one-way connection. A message will be sent off to the IMP system on the remote machine, but `ImpLocate()` will return immediately to its caller without waiting for a reply. The calling program is now in a 'waiting for connection' state (in fact these tasks are often best thought of a 'state machines' driven by external message events).

2. Eventually, an IMP system message message reporting on the locate attempt will be delivered to the task. This should be passed to `ImpSystemMessage()` which will handle any internal IMP requirements for the message and will extract the task name and the status of the locate attempt which the calling task can now check. If the locate attempt was successful, the task can now call `ImpConnect()` in an attempt to establish a connection with the task. `ImpConnect()` has a 'Wait' argument which it sets if the connection cannot be established immediately, and for this network connection this will be returned set true. The task is now in a 'waiting for connection' state.

3. Eventually, a second IMP system message will be received giving the status of the connection attempt. Again, this should be analysed by `ImpSystemMessage()`. If the attempt was successful, the task is now in a 'connected' state and can start sending messages to the remote task.

The event-driven nature of a task using IMP can be illustrated just by providing a list of the system messages that it might encounter. Note that IMP makes its internal operations open to the world, rather as the Pompidou center in Paris has its plumbing on the outside. You may not care to see all this going on, but there may be times you will find it useful. It would have been possible to isolate the application program from all this, using an interface based on callbacks when messages of specified types were received, but this is usually done at a slightly higher level than IMP. (IMP is not usually called directly by applications programs: at AAO it is usually wrapped up as part of the DRAMA system which hides the IMP goings-on from the applications programmer.)

System messages can have any of the following codes, some of which are handled entirely by `ImpSystemMessage()`, some of which require additional action to be taken by the receiving task, and some that provide information the task may need to take notice of:

IMP_LOCATE_REQ is the message that comes back to a task that called `ImpNetLocate()`, indicating whether or not the required task has been located.

IMP_CONNECT_REQ is the message that comes back to a task that has called `ImpConnect()` in a way that required that it wait for the connection (either a connection to a remote task or a 2-way connection that has to be accepted by the target task). This indicates that the connection has either been completed or has been rejected.

IMP_SYS_DISC is a message sent by a task as it closes down to all the tasks that have been connected to it.

IMP_SYS_CRASH indicates that a task with which the current task has been in communication with has crashed. When a task crashes this message is sent by the IMP 'Master' task to all tasks that the crashed task had a record of communication with.

IMP_SYS_REQUEST is a request from another task that it be notified when our task is able to receive messages from it again. This message, and the sending of the requested notification, is handled automatically by the IMP system and the main task code can ignore it. This is actually the message sent as the result of a call to `ImpRequestNotify()`.

IMP_SYS_NOTIFY is the message sent in response to a call to `ImpRequestNotify()`. When it receives this, our task can start sending messages again to the target task.

IMP_SYS_REPORT is a message sent as part of the internal flow control protocol. It indicates that a target task has reported an empty buffer, but there are still messages sent on the connection in question that have not been processed by the target task. This message is handled completely by the IMP system and should be ignored by the main task code.

IMP_SYS_SYNCH is another message sent as part of the internal flow control protocol. It is sent when a connection is completely clear,and is handled completely by the IMP system and should be ignored by the main task code.

IMP_SYS_CONNECT is a request from another task that the receiving task initiate a two-way connection with it. The receiving task must call `ImpAccept()` to explicitly either accept or reject the connection.

IMP_SYS_SHUT is a request from another task that the receiving task shut itself down. This is provided in order to provide an emergency clean shut-down mechanism for programs. (Sending this message, using a utility routine, is a cleaner alternative to using the operating system to kill the program, assuming it is still in a position to accept it.)

IMP_SYS_SEND_ERR indicates that a message previously sent by the receiving task failed to be delivered properly. Normally, this only happens in the case of network messages, since a failure to send a message to a local task will be reported immediately by `ImpSend()`. Because of delays in the system, if a series of messages is being sent without each being acknowledged individually, it may not always be obvious which message failed to be delivered.

IMP_SYS_MACHINE_LOST indicates that contact with a remote machine has been lost by the networking part of IMP. This message is sent to all tasks on any machine that was in contact with the lost machine. A task that receives this message should close down any connections it may have with that task, using the routine ImpLostMachine().

IMP_SYS_LOAD_ERR indicates that a task load request issued by the current task (through a call to `ImpRunTask()`) has failed.

IMP_SYS_TASK_EXIT indicates that a task whose loading was requested originally by the current task (through a call to `ImpRunTask()`) has exited.

IMP_SYS_LOADED indicates that a task whose loading was requested originally by the current task (through a call to `ImpRunTask()`) has successfully called `ImpRegister()` and registered itself with the IMP system.

IMP_SYS_CONN_CLOSE indicates that a task has broken off a connection with the current task and has requested that any necessary housekeeping be performed. This housekeeping will be performed by `ImpSystemMessage()`.

IMP_SYS_ENQ_HANDLE is only sent to a 'Translator' task and indicates that a task is enquiring whether or not a named task is handled by the translator task. The task must respond with a call to `ImpHandled` to indicate whether it does or not.

IMP_SYS_REJECT indicates that a task has rejected a connection that the current task attempted to make with it.

IMP_SYS_REJECT is sent when the current task has requested a connection with another task and that other task has rejected the connection. `ImpSystemMessage()` will handle any necessary housekeeping for the message, but the calling program will need to note that the connection has been refused.

IMP_SYS_NOTE_EXIT indicates that a task on a remote machine has exited. This message is normally only sent to the IMP network tasks, and an ordinary task should ignore it.

IMP_SYS_NET_PRIVATE is an internal message, usually connected with flow control, sent between network tasks on different machines. A normal task should never get such a message, and should ignore it if it does.

IMP_SYS_BULK_DATA indicates that bulk data in a shared memory section is available. The task should map the shared memory using `ImpHandleBulk()` and should eventually indicate that it has finished using the data through a call to `ImpBulkReport()`.

IMP_SYS_BULK_WAITING indicates that another task (usually a network task) has bulk data available for the current task, and needs to know where to deliver it. The task should create a suitable shared memory section and should indicate that it is ready for the data through a call to `ImpReadBulk()`.

IMP_SYS_BULK_TRANSFERRED indicates that a certain fraction of a bulk data transfer has completed. The receiving task need not do anything. However, a sending task may want to take note of the `Released` flag in the message, since this indicates it is now free to modify or delete the original data. A target task (one to which the bulk data is being sent) may need to monitor the percentage transferred, particularly if it is trying to make use of the data as it is being sent.

IMP_SYS_BULK_INTERNAL is an internal handshaking message used by the bulk data routines. It should be ignored by an ordinary application — any processing needed will have been carried out by `ImpSystemMessage()`.

IMP_SYS_MACH_DIED is an internal message sent only to the IMP networking tasks when contact with an external machine is lost. An ordinary IMP program should never get such a message. If it does, it should ignore it.

IMP_SYS_CONNECTED is an internal handshake message sent between IMP network tasks. An ordinary IMP program should never get such a message. If it does, it should ignore it.

It is hoped that by making these possible conditions so explicit, and by insisting that any task be expected to handle all of these possible conditions, some additional reliability may be achieved. For example, most programs may be reluctant to consider the possibility that a remote machine may crash, and may prefer to ignore it; by insisting that a program be prepared to handle a message of type IMP_SYS_MACHINE_LOST this design may encourage the production of code that survives such a case.

Note that new system messages tend to appear with new versions of IMP, as new facilities appear. Code using IMP should be written to ignore system messages it doesn't expect, rather than to test for all the types known at the time of writing and outputting 'unexpected system message' errors if a new message type is seen.

6 Handshaking protocols

Often the obvious protocol to use when sending messages to another task is a 'send-acknowledge' protocol, where a task sends a single message to another task and then waits for an acknowledgement from that task. In some cases, this imposes an unnecessary delay, especially if the sending task does not actually need any particular information back from the receiving task. However, if the sender just keeps blasting messages at the receiver with no protocol at all, eventually the receiver may be unable to keep up with the sender. Under these circumstances some protocol is needed where the sender can pause until the receiver indicates that it is ready to accept more input.

IMP provides 'flow controlled' connections to handle this requirement. When a connection is created (either through a call to `ImpConnect()` or through a call to `ImpAcceptConnect()`) the flag 'IMP_FLOW_CONTROL' may be specified. In this case, a flow controlled connection is created. When messages are sent on such a connection, IMP sets up a mechanism whereby the sending end monitors the amount of data sent through the connection and the receiving end periodically reports on the amount of data that has been handled by the target task.

Note that the distance between sending and receiving tasks (which may be on different continents!) will mean that there can be a noticeable delay between the target task handling a message and the sending task discovering that it has done so. During this delay, the sending task may well have sent a number of additional messages. This means that there are often a number of messages that have been sent but which the sending task does not yet know to have been handled by the target task. If the number of messages in this 'limbo' state aproaches the capacity of the buffers declared for the connection, the IMP system in the sending task will refuse to send any further messages until it knows that the backlog has been cleared.

When this condition applies, a call to `ImpSend()` or `ImpSendPtr()` will fail with an `IMP__NEED_-SYNCH` status. Alternatively, a call to `ImpCheckSend()` to see if a message can be sent will return the same status. Once this condition is detected, the sending task should not send any more messages on this connection until it is notified that the connection is completely clear. What the sending task should do is call `ImpRequestNotify()` to trigger the required notification. The notification takes the form of a system message which `ImpSystemMessage()` will classify as being of type IMP_SYS_NOTIFY. `ImpSystemMessage()`, also returns (in the `InputNumber` field of a system information structure) the number of the connection which has cleared. Once it has made the call to `ImpRequestNotify()` the sending task should not attempt to send messages using the connection in question until it gets the IMP_SYS_NOTIFY message.

If the sending and target tasks use two-way connections to handle their own synchronisation, waiting for a message to be acknowleged before sending the next, flow controlled connections represent an unnecessary overhead. The flow control protocol is — needs to be — very conservative. The 'IMP__ NEED_SYNCH' status indicates that the system cannot *guarantee* that a message can be delivered, although in most cases there will be no problem. As a result the sequence of calling `ImpRequestNotify()` and the subsequent wait for the IMP_SYS_NOTIFY message will sometimes be invoked even though the buffers are in fact clear. (The system simply doesn't *know* they are clear.) Having to handle this protocol as well as their own private message acknowlegements will complicate the code for such a pair of tasks. However, where the tasks involved do not provide this synchronisation themselves — for example, if a one-way connection is being used to continually send some sort of monitoring information — then a flow controlled connection should be used.

Flow control can be specified for both remote and local connections. However, it is not *needed* for local connections, where information about the rate at which the target task is handling its messages is available directly to the sending task. Since there is only one message buffer involved in a local connection, a call to `ImpSend()` or `ImpSendPtr()` that returns a status of 'IMP__NO_SPACE' indicates that the target task is not keeping up with the flow of messages. In this case, the same handshaking protocol as for a flow-controlled connection can be used. The sending task can call `ImpRequestNotify()` and wait for an 'IMP_SYS_NOTIFY' message to indicate that the connection has cleared. While this is slightly more efficient than using the full flow control protocol, the additional complexity of having to ensure that the connection is actually to a local task probably means that in most cases using flow control is the better option. `ImpCheckSend()` can be used for both flow-controlled connections and local uncontrolled connections — it classifies both the 'no space' and the 'need synch' codes as transient conditions that should be handled by calling `ImpRequestNotify()`.

Some additional points about flow-controlled connections:

- Using `ImpCheckSend()` simplifies the coding of the decision as to whether or not an error condition is serious or just a transient condition that can be handled through a call to `Imp-RequestNotify()`. However, it has the additional attraction that, as an enquiry routine, if it finds that the message in question cannot be sent it merely reports this and does not generate error messages. You can choose to use tentative calls to the sending routines to find out if there is a problem or not, and this is slightly more efficient — the buffer sizes do not need to be checked twice, once by the check and once by the send call — but you will then have to control the error messages generated by the send routine if it detects an error condition. You can do this either by using Ers routine calls to annul the error status, or you can specify the 'IMP_NO_FULL_MSG' flag when creating the connection.
- The routine `ImpAcceptConnect()`, which should be used if you want to specify flow control for the second half of a two-way connection, is the same as the older `ImpAccept()` routine except for the addition of a 'flags' argument provided specifically to enable the flow control flag to be specified.
- If you find that you cannot send a large message on a connection, you may still be able to send a smaller one. There may be cases where this is a sensible thing to do, and it is perfectly OK. However, once you have called `ImpRequestNotify()` for a given connection, you should not attempt to send more messages on that connection until you get the 'IMP_SYS_NOTIFY' message indicating that the connection is clear. (This is one reason the system does not automatically call `ImpRequestNotify()` when it cannot send a message on a connection — it prefers to leave open the option of sending a smaller message instead.)
- Oddly enough, it is not completely pointless to use a call to `ImpCheckSend()` and to call `ImpRequestNotify()` to trigger notification messages even when using a non-flow controlled remote connection. In this case, the system will not be able to tell the sending task anything about the rate at which the target task on the remote machine is handling its messages (you need a flow controlled connection for that) and so you may still get error messages indicating a delivery failure at the remote end of the connection. However, this will pick up the case where the IMP networking is the limiting factor; you will get an 'IMP__NO_SPACE' error if the 'Transmitter' network task cannot keep up with the sending task, and the `ImpRequest Notify()` call will trigger your being notified when 'Transmitter' is able to send on your message. However, if you are going to do this, then you would probably be better off using a flow-controlled connection.
- Flow controlled connections were introduced after IMP version 0.9. Previous versions had included the use of `ImpRequest Notify()` to handle the flow control problem, but only for

local tasks, and had promised the introduction of a ‘synchronise/acknowlege’ protocol for use with remote tasks. It was eventually realised that the use of flow controlled connections allowed essentially the same protocol to be used for both local and remote tasks and so this has superseded the proposed ‘synchronise/acknowlege’ protocol.

7 'Peeking' at messages

Normally, the intention is that programs using IMP will be event- driven, able to respond to any type of message as it arrives and so able to handle all messages in the order in which they are delivered. While most tasks written specifically to use IMP should be able to achieve this, there are some cases where this will be a particularly awkward requirement.

Consider two separate, although related, examples. In the first, a task is running that accepts a command that causes it to dive off into some very complex and long-winded calculation. It is often awkward to code such things so that they can easily keep returning to a main node that can handle any message and then continue the calculation where it was interrupted. However, we do need the process to be responsive to the arrival of messages, if for no other reason than that we want it to be able to respond to a request to cancel the current calculation. It is often possible to insert calls to `ImpMessageCount()` at strategic parts of the calculation, and so return to a main, message handling, node only when a message has actually arrived, but even so this means that the calculation will be interrupted by the arrival of *any* message. Given that a message may be nothing more than a status request or something even more trivial that the process would much rather ignore at this point, this is an awkward way to operate. Ideally, the process needs some call that has the effect of asking 'have any messages arrived that I'm particularly interested in at the moment?'.

The other example involves a task that is determined to issue a prompt and wait for a reply. While one might not write a task like this from scratch, one might well be trying to run an existing task under a new, IMP- based, environment, and it may be difficult to reconstruct such a task to follow the preferred IMP model. It should be easy to have the task send its prompt as an IMP message, and to get its reply back as an IMP message. The problem is that it may send off its prompt and then get some quite unexpected message back while it is waiting for the reply. It may, for example, get an IMP system message telling it that the task it is expecting to provide the reply has crashed! Or it may get a cancel request, or some completely unrelated message. Such a task could fall back on a strategy of reading all messages that arrive and ignoring any that it cannot handle at that point, dealing only with ones of immediate interest (probably just the expected reply, perhaps including messages indicating any problems with the replying task). But it is a shame to cast messages aside simply because they can't be handled there and then. You could imagine the task queuing them itself and dealing with them once it finally gets back to some 'main node', but this would involve a great deal of effort. What this task needs is some way of asking 'can I have a look at just the messages I'm interested in at the moment?'

The solution provided by the IMP package is to allow 'peeking' at messages. A 'peek' is a read of a message that leaves the message still in the message queues. Think of it as a 'non-destructive' read. A read call (to `ImpReadPtr()` or `ImpRead()`) can specify a 'peek' option bit in the flags that it passes to the call. A read call that sets the peek flag will return the next message that has not yet been either read or peeked at. So you can read a message twice, once by peeking at it and then by reading it normally. The reasoning behind this is that really only the program itself knows whether a message is one it can handle at any given point, and the only way to let it find out is to let it look at the message in some non-committal way. A message that has been peeked at can be set as 'read' by a call to `ImpSetAsRead()`, and this effectively removes it from the system. In particular, if a program that is peeking at messages encounters a system message, it might be well advised to process it immediately (by calling `ImpSystemMessage()`) since system messages shouldn't normally be delayed unduly. However, it is important not to process system messages twice — that could be most confusing — so any system message processed after a peek should always be set as 'read'.

Using this facility, the first example task (the one that wants to know whether to abort what it is doing to process a new message) could see if any messages have arrived and peek at any new messages. As a result of this peek, it could decide whether or not to return to its main node and read them normally.

These second example task (the one that is waiting for a specific reply) can wait for the next message, then peek at it, see if it is what it wants and if not can just wait for the next message, which it can handle in the same way. It should handle system messages as them come in, in case they refer to the task it was communicating with (a task crashed message, for example, would be a good one to look closely at).

A simple extension to this will allow a call to reset all the 'peeked at' flags, so a task could work again through the messages it has already peeked it. This is simple to implement but has not been required yet.

Tasks that peek at messages should realise that such messages stay in the message buffers, so they should perhaps allocate larger message buffers than they would otherwise have specified.

8 Task loading

IMP provides the routine `ImpRunTask()` which allows a process registered with IMP to request that a new process be started up on a specified machine. The process making this request then becomes the 'requesting process' for this new task, and as such is kept informed of its progress. In particular, the requesting process will be notified a) if the process fails to load properly, b) if the new process registers with the IMP system (by calling `ImpRegister()`), and c) when the new process terminates, either normally or as the result of a crash.

The interface to `ImpRunTask()` is made complicated by the fact that process creation differs so much from machine to machine. The intention has been to try to provide a basic interface that will work well on all the supported systems, but which allows system-specific control of a task as well. In most cases, the program that issues a call to `ImpRunTask()` will know what type of machine the target system is, so it is not unreasonable to make system-dependent features available, but at the same time the more general the interface the better. The call looks as follows:

```
ImpRunTask
(ProcID,Machine,TaskName,ArgString,Flags,TaskParams,Status)
```

Here, 'ProcID' is the usual IMP identifier for the calling task and 'Machine' specifies the machine to be used (a null argument indicating the local machine). The basic arguments that specify the task and how it is to be run are 'TaskName' and 'ArgString', both of which are character strings. More detailed control can be exercised through the use of 'Flags' and 'TaskParams', which tend to be system-dependent. 'Status' is the usual IMP inherited status argument.

'TaskName' specifies the name of the task to run. In systems like VMS and UNIX, this will be a file name — the name of an executable program. In systems like VxWorks where tasks are all linked together and what is really wanted is the name of a subroutine, this needs to be something that can be processed to give a subroutine address. In all cases, it is possible for a simple name, like 'testprog' to be accepted. If the type of system running the program is known, a more explicit program name can be given, such as 'disk$user:[ks]testprog' (for VMS) or '/home/aaossc/ks/testprog' (for UNIX).

'ArgString' is a character string giving arguments to be passed to the task being loaded. In cases where what is loaded is a process that registers with IMP and takes commands through IMP messages, it will be best to control the task in this way. However, if the task is a general purpose (non- IMP) program, (or even if it does use IMP, in some cases) it will be convenient to be able to pass it parameters. All the IMP target systems to date allow the passing of such parameters, although in subtly different ways. IMP tries to make sure these arguments are interpreted as far as possible in the way they would be interpreted if typed into the usual command processor for the system in question. Fortunately, most command processors have a similar syntax, and an argument string with blank-separated strings and numbers like "fred 10 500" can be handled by most systems.

'Flags' is a bit mask that can be used to trigger system-specific control over the task, usually in ways that take additional parameters from the 'TaskParams' structure. For example, if the 'absolute priority' bit in 'Flags' is set, then IMP will (if the target systems permits, which all do to date) set the priority of the loaded task to the integer value given by TaskParams.Priority. However, at this point you really do need to know the nature of the target system — a value of 128 will give you an average priority task under VxWorks, will be invalid under VMS and will be clipped at 32, which will be the highest priority task on the system (although in practice you probably won't have the privilege to run such a task) and, depending on the actual UNIX system, will give you a very low priority task under UNIX. IMP assumes that if you use this bit you know what you're doing! Flags that are not supported by a given implementation are ignored by it.

The minimum syntax for 'TaskName' that will be accepted on all platforms is a single name, such as 'myprog'. This is interpreted as follows by the various current implementations:

VMS This is the name of a program (a .EXE file) in the current default directory of the IMP 'Master' task, or in the directory from which the IMP 'Master' task was run. Alternatively, it can be a logical name (known to the IMP 'Master' task) that translates to the full file name of a program (a .EXE file). Also, it may be a defined symbol known to the IMP 'Master' task that causes a program to be run, or may be the name of a command that can be spawned. (So it can, for example, be 'dir' or 'copy' which could be spawned as commands.)

UNIX This is the name of a program file that can be found in the execution path of the IMP 'Master' task. Alternatively, it may be the name of an environment variable known to the IMP 'Master' task that translates to the full file name of a program file.

VxWorks This is the name of a subroutine already loaded and whose name is in the system symbol table. (A more complex interface allows IMP programs running under VxWorks to associate symbols with function addresses, and this can be used in cases where there is no system symbol table available.)

The interface also allows an extended syntax for 'TaskName' that will have some meaning on all platforms. This is 'location:program' where "location" is a name that can be symbolically defined as the location of a program, or will be ignored by systems where this is meaningless, and "program" is a program name. This is interpreted as follows by the various implementations so far.

VMS "location" is a logical name known to the IMP 'Master' task which translates to the name of a directory where there is a program file (a .EXE file) called "program".

UNIX "location" is an environment variable known to the IMP 'Master' task which translates to the name of a directory where there is an executable file called "program".

VxWorks "location" is normally ignored, and "program" is the name of a subroutine already loaded and whose name is in the system symbol table. Alternatively, the more complex interface already mentioned above will allow the whole "location:program" string to be defined as a symbol that is associated with the address of a program already loaded.

The implementations also allow the following values for 'TaskName', but these are system-dependent and should be avoided if possible.

VMS 'TaskName' can be a full file name, such as 'disk$user:[ks]program', the default file extension being '.EXE'.

UNIX 'TaskName' can be a full file name, such as '/home/aaossc/ks/program'.

VxWorks No specific other syntax is accepted at present, although the more complex interface mentioned above will allow any string at all to be defined as a symbol associated with an address.

The minimum way of specifying 'ArgString' that is accepted on all systems is as a null argument — either an actual NULL value, or an empty string — one that begins with an end-of-string zero byte. In this case, none of the implementations will pass any arguments to the loaded task. The implementations treat actual values of ArgString as described below. This looks complex, described formally like this, but it is hoped that in fact it all just works as you'd expect from using the various systems.

VMS If an argument string is specified, then 'TaskName' must have specified the name of a symbol that will cause a program to be run. IMP will append 'ArgString' onto 'TaskName' (with a separating space) and will spawn the resulting command. So, for example, if 'TaskName' is "dir" and 'ArgString' is "*.c", the result will be the spawning of a "dir *.c" command. If the program spawned is a user-written one, it has to have been defined as a symbol and it has to have been written to pick up its arguments from the command line — a C program can do this using the traditional `main(argc,argv)` main program, one written in Fortran or some other language will have to use LIB$GET_COMMAND or the CLI library.)

UNIX If an argument string is specified, it is parsed into a set of separate character strings, delimited by one or more spaces. If a string containing a space is to be treated as a single string it should be enclosed in either 'single' or "double" quote marks. The new process is forked and the specified program is executed with the program name and the addresses of these separate string arguments passed as program arguments, to be picked up, for example, by a C program using the traditional `main(argc,argv)` main program. This mimics the behaviour of most UNIX shells when starting a new process.

VxWorks If an argument string is specified, it is parsed into a number of separate arguments separated by one or more spaces or commas. If a space or comma is to appear in a string, the string should be enclosed in either 'single' or "double" quote marks. A set of arguments for the called subroutine are then created from these strings as follows: if a string can be decoded into a floating point number or an integer (the test being that the former contains a decimal point, so "9." is a floating point 9.0 and "9" is an integer 9) then the numerical value itself forms the argument; otherwise, the address of the string is used. This mimics the behaviour of the standard VxWorks shell, except that the use of spaces to delimit arguments has been added to allow compatability with VMS and UNIX.

For a full description of the use of 'Flags' and 'TaskParams' see the system-independent description in the appendix of `ImpRunTask()`. The various implementations to date support the following features:

VMS The flags IMP_REL_PRIO, IMP_ABS_PRIO, IMP_SYMBOL, IMP_PROC_NAME, IMP_PROG and IMP_NAMES and the fields in TaskParams that are associated with these are supported. All other flags are ignored.

UNIX The flags IMP_REL_PRIO and IMP_ABS_ PRIO are supported, together with the associated TaskParams.Priority field; all others are ignored.

VxWorks The flags IMP_REL_PRIO, IMP_ABS_ PRIO, IMP_PROC_NAME, IMP_SET_-BYTES and the fields in TaskParams that are associated with these are supported. All other flags are ignored.

Some systems support the concept of a process name (VMS, VxWorks, for example), while others (notably UNIX) do not. On systems which do not, the IMP_PROC_NAME flag and the

TaskParams.ProcName field are ignored. On systems that do support the concept, a unique name will be generated by the system if the IMP_PROC_NAME flag is not set.

A task that calls `ImpRunTask()` will always receive either a system message of type IMP_SYS_-LOAD_ERR to indicate that the load of the new task failed, or a message of type IMP_SYS_-TASK_ EXIT to indicate that the task started and has exited (either voluntarily or as the result of crashing or being killed by another task). If the loaded task calls `ImpRegister()` successfully, then the requesting task will receive a message of type IMP_SYS_LOADED when this happens, indicating that it should now be able to locate and communicate with the new task through IMP. (Of course, if something goes seriously wrong, there may instead be a message of type IMP_SYS_SEND_ERR or IMP_SYS_ MACHINE_LOST that can be connected with the load request. A truly robust task will allow for these as well.)

The example IMP utility program "impload", which forms part of the IMP code, demonstrates the use of IMP task loading.

9 Status conventions and error reporting

IMP uses a convention referred to as ‘inherited status’. In almost all user-callable routines, the last argument is a pointer to an integer status code. If this integer is non-zero (which indicates an error, in this convention) when the routine is called, it will return immediately. If it is zero when the routine is called, the routine will execute and will set the status integer to a non-zero error code if an error occurs. If the routine executes normally, the error code will be left with its original zero value.

This convention is useful because it means that a long sequence of calls to IMP routines can be made without having to check the status after each call to decide whether or not it is safe to call the next routine. If there was an error in one routine, the inherited status convention ensures that subsequent IMP routines will not do anything at all and so can be called safely. It has the disadvantage that if you do not test the status after each call you don’t always know just which routine generated the error. The error codes themselves are all given symbolic names that begin with ‘IMP`__`’ (IMP followed by two underscores) in the ‘imp.h’ include file.

IMP uses the Ers error reporting system to log error messages. The Ers system is part of the AAO’s DRAMA data acquisition environment, and is described in detail in the DRAMA document “Drama Error Reporting System (Ers)” by Tony Farrell. IMP routines that return bad status values always make calls to the error reporting routine `ErsRep` when doing so. This generates an error message but this is not output immediately. Instead the Ers system queues the error in a stack of messages. This allows a calling routine that decides that it can handle the error internally and that the user therefore need never see the error message (and should not, since it would be confusing!) can use the Ers system to annul the message and ensure that it is never output. Most programs that use IMP will not need to do this, but writers of such programs need to be aware of the use of Ers by IMP.

One consequence of IMP’s use of Ers is that error messages generated by IMP, since they are queued by Ers, are not always going to be output as they occur and so can end up out of step with other messages output directly using printf(), for example. Another consequence is that IMP programs need to be linked with the Ers library. This is available from AAO, but can be tricky to build since it makes use of a deal of the DRAMA infrastructure. As an alternative, IMP provides its own stand-alone implementation of Ers (using code originally written by Allan Brighton of ESO) which can be used instead.

The type of the status variable needs a mention. It is always declared as being of type ‘`IMP_Status`’, which is defined in ‘imp.h’. It is always a signed integer, and it is always declared as the smallest integer type that can hold a 32 bit value. This means that it is a long int on most 32 bit systems, but is an int (not a long int) on an OSF/1 Alpha system where a long int is 64 bits. At first sight it might seem more sensible to make it a long int on all systems, but there are two reasons for not doing so. One is that in some cases this value has to be sent from one machine to another, so it has to be defined as the same length on all systems, and the other is that the type chosen has to be compatible with the standard DRAMA status type (defined in the DRAMA include files as type ‘`StatusType`’, and this is constrained by a requirement that it match a FORTRAN INTEGER, which even on an Alpha is a 32 bit quantity. (This is so that DRAMA routines are compatible with routines used from FORTRAN.)

So long as you always include the ‘imp.h’ include file — and you have to, or all the other types you use won’t be defined — prototypes for all the IMP routines will be provided for your compiler and any mismatch in types can be picked up. (So long as your compiler supports function prototypes, but most do nowadays.)

10 Systems with multiple IP addresses

IMP 1.4.2 added support for systems with more than one IP address. You may, for example, have two machines on a general-purpose network that need to transfer data from one to the other very efficiently and so have a private ethernet connection between them. These machines will then have one IP address that specifies their connection to the common network and one IP address that specified their connection to the private connection. Symbolic names can be defined for the two IP addresses.

You might, say, have a machine called 'fred'. This would have a number of possible symbolic names: its full symbolic name, 'fred.aao.gov.au', and possibly a number of alternative names that all translate to the same numeric IP address — its address on the general-purpose network. But 'fred' might also have a second interface, with a separate IP address, and the symbolic name 'fred2' that translated to that second IP address. (And, of course, there could be a set of alternative symbolic names that also translate to that second IP address, such as 'fred.aao.gov.au'.)

Not all machines would have access to that second interface, but on a machine that did, any connection set up to 'fred2' should use that second, dedicated, interface. Such a connection could be set up by 'telnet', or 'ftp', or, naturally, IMP.

In the main, IMP handles such systems transparently. You can call `ImpNetLocate()` giving the machine name as 'fred2' and so can locate a task on that machine and can then set up a connection to it. That connection will use the dedicated interface. There are, however, a couple of things to watch:

Task names Once you locate a task called, say, 'target' on the machine specified by the name 'fred2', this task will become registered on the local machine. It will be registered as 'target' and will appear as a task on the machine 'fred2'. That's what you'd expect. Note, however, that if you then use `ImpConnect` to set up a new connection and specify the target task by name as 'target', you will automatically use the dedicated link — since 'target' is registered as being on 'fred2'. If you particularly want a second connection that uses the common link, you can set it up, but you have to omit the taskname in the call to `ImpConnect` and supply an IMP_TaskID structure that specifies the task explicitly by process ID and machine IP address. (A second call to `ImpNetLocate()` giving the machine name as 'fred2' will return a suitable TaskID structure.)

Startup files IMP needs to know about such multiple IP address machines. This is because IMP may set up a connection to 'fred2' and some time later the networking tasks may discover that 'fred' has gone down. IMP needs to know that this implies that 'fred2' has also gone down. There are other similar scenarios. IMP is told about such IP 'aliases' through its startup files. In this case, all machines using 'fred2' should have the following line in their startup files:

```
Note alias fred2 for fred
```

Version 1.4.4 added a new feature to get around some problems found on dual IP address systems. It may be that you have a machine with more than one IP address, but some other machines can access it only through one of these addresses and others can only access it through one of its other IP addresses. This sort of system can be set up for security reasons, or for example to have a cluster of machines invisible to the outside world other than through an 'interface' machine. In such a system the private cluster machines can only talk to the interface machine using their 'private' IP address for it, and all other machines can only talk to the interface machine using its 'external' IP address.

The problem is that IMP puts a 'from' IP address in all its messages, and this 'from' address is the same for all messages sent from that machine, private or external. (Under UNIX the address used is that corresponding to the machine name returned by a '`uname()`' call.) This means that some machines, either the private machines or any external machines, get IMP messages that seem to come from a machine that they can't access! To overcome this problem, IMP now supports a further startup option, of the form:

```
   Use address addr1 for addr2
```

The effect of this message is that whenever a machine has to send a message to the IP address given in addr2 it will actually do the sending using the IP address given in addr1. In the example of the 'interface' machine given above, if the interface machine puts its internal IP address into all its messages, then an external machine will need this startup option specified in its startup file. In this case, 'addr2' will be the internal IP address of the interface machine, and 'addr1' will be the external IP address.

Note that 'addr2' in this case will almost certainly have to be given in dot notation, which is a pity, but the external machine almost certainly won't have the interface machine's 'private' IP address in its tables — if it did there wouldn't be a problem! Note also that if you have a 'use address' option in a startup file you will almost certainly also want to have a 'note alias' option giving the same two addresses. (IMP doesn't assume that 'use address' implies an alias, although it almost certainly does in practice.)

11 Shared Memory Sections in IMP

All systems on which IMP is implemented support the concept of a 'shared memory section': an area of memory that can be accessed by more than one task, providing a very fast means of communicating data between tasks. The implementation differs considerably from system to system, and some systems provide more than one mechanism for achieving the same effect. For example, UNIX systems can create 'System V shared memory sections' using `shmget()` calls, or can use the `mmap()` call to allow different tasks to map the same file. At the other extreme, in systems such as VxWorks all tasks share the same address space, and all memory is shared between all tasks on the system.

IMP uses shared memory sections internally for communication between tasks on the same machine, although the details are, of course, hidden from the user. IMP version 1.1 introduced facilities that support the explicit use of shared memory sections for efficient handling of large amounts of data with minimal overheads. A program can create a shared memory section and then use IMP to notify another task on the same machine of its existence so that the other task can access the data in the shared memory section. This is a fairly straightforward process. More interestingly, IMP also provides facilities for transferring data directly from a shared memory section on one machine to a shared memory section on another machine. The routines in question form the 'bulk data transfer' facilities of IMP, since they are intended for use mainly with very large amounts of data (tens of Megabytes or more). These are described in more detail in the section 'IMP Bulk Data Transfer Routines'.

When a calling program makes use of a shared memory section it is often convenient for it to create this outside the IMP system and then describe it to IMP, rather than let IMP create a memory section for it. This allows the program full control over the details of the shared memory, which can be useful if a program is subject to some external constraints on its use of shared memory — for example, part of a data acquisition system may be dealing with data placed in shared memory by some detector control package. Under UNIX, the data may already be in a specified file that can be accessed most efficiently using `mmap()`. A display package may be able to display data placed in shared memory if it is of a type that it specifies. For these reasons the routine `ImpDefineShared()` allows an externally defined shared memory section to be described to IMP.

`ImpDefineShared()` takes a system-dependent description of a shared memory section and initialises an IMP shared memory description structure of type `IMP_SharedMemInfo`. This is a general purpose structure that can accomodate all the vagaries of shared memory specifications of the systems that IMP supports. This structure can then be passed to other IMP routines such as `ImpSendBulk()` that will arrange for its contents to be accessed by other tasks. Note that if you create a shared memory section that you want IMP to access, you must call `ImpDefineShared()` to set up the `IMP_SharedMemInfo` structure that describes it — you should not just set the fields in the structure directly. This allows additional fields to be introduced into the structure later without breaking existing code, since `ImpDefineShared()` will always initialise all the fields of the structure to sensible defaults. It also conveniently bypasses the fact that for portability reasons the fields in such a structure are not defined all that straightforwardly.

It is possible to set a flag in the call to `ImpDefineShared()` that requests that routine to do the actual creation of the shared memory section; this can simplify code that merely wants a large amount of sharable memory and is not having to use an externally created memory section. If you use `ImpDefineShared()` to create the memory section, you can use `ImpReleaseShared()` to unmap it (ie to detach the program from it, so the program's address space is no longer mapped onto the data in the shared memory section) and optionally flag the shared memory section for deletion once no task is atttached to it. Generally, it makes most sense if only the task that originally created the section sets this deletion flag. If you have no interest in the details of the shared memory section created, you can call `ImpDefineShared()` with a flag that will just create a temporary memory section of the required size — the advantage of this is that it allows the call to be completely system-independent.

12 IMP Bulk Data Transfer Routines

This section describes in some detail the use of the IMP bulk data handling routines. Because it is detailed, it may look complicated. However, the code required to follow the sequences shown is usually only a few lines of C, since the IMP routines package up most of the necessary work.

The IMP bulk transfer routines — `ImpSendBulk()`, `ImpReadBulk()` and the associated routines such as `ImpHandleBulk()` and `ImpReportBulk()` — are designed to transfer data held in shared memory sections as efficiently as possible, either to tasks on the same machine or to tasks on remote machines. When the tasks are on the same machine this is mainly a case of making the target task aware of the existence of the shared memory. When the tasks are on different machines, this is more complex and involves the transfer by the IMP network system directly from the original shared memory section into a shared memory section on the target machine that has to be created by the target task.

Since an IMP message can be of any length, it is possible to handle even large amounts of bulk data (the tens of megabytes or more that the bulk data routines were designed for) using a normal IMP connection. However, there are two problems with the normal IMP connections that make it worthwhile having a separate set of routines explicitly designed to handle bulk data. The sizes of IMP connections have to be specified in advance, meaning that one has to allocate memory for the worst case. It would be possible to set up a new connection for each large transfer, but apart from the (relatively slight) overhead of doing this, the buffers for normal IMP connections have to come from the memory specified by a task when it first registers with IMP, meaning that this initial memory specification has to be a worst-case specification. This is not fundamental; the extension of IMP to allow dynamic allocation of memory for connections has been planned ever since the first release. However, even if a new connection could be allocated for each bulk data transfer and this could be done without pre- allocation of memory, there are still problems connected with the need for data copying. A call to `ImpRead()` specifies a message buffer that is outside the IMP system and so needs data to be copied into it. This is a significant overhead if it means two buffers of this size, one inside IMP and one outside, have to be allocated and the data copied between them. `ImpReadPtr()` does not have quite this data copying problem, but it uses memory that is allocated from within IMP, and this is not always most convenient for a task that may want to read directly into some display buffer or even into a mapped file. None of these are significant problems for small messages, but for seriously large messages they can be serious limitations.

Enter the IMP bulk data handling routines. These allow direct transfer of huge amounts of data from shared memory sections that are — as far as possible within the constraint that they have to be in shared memory — under the control of the sending and target task. The following sections describe their use in detail.

12.1 Sending bulk data

The basic sending mechanism is simple. A task creates a shared memory section in whatever way it prefers. Under UNIX this can System V shared memory or a memory mapped file; under VMS it is a global section; under VxWorks it is any area of memory, since all memory is shared. The task calls `ImpDefineShared()` which sets up an IMP shared memory description structure (of type `IMP_SharedMemInfo)` containing sufficient information for another task to access the shared memory. The task then passes this description structure to `ImpSendBulk()`, specifying a connection already made to the target task. This connection, which is used only for the handshaking messages associated with the transfer, needs to be a two-way connection with messages of at least a specified size (given by the constant IMP_BULK_MESSAGE_LENGTH). This is all that is required to initiate the transfer of the data.

This of course leaves the task with a shared memory section mapped and expecting that some other task will also map that section and use the data in it. If the target task is on the local machine, it will be the target task itself that does this mapping. If the target task is on a remote machine then it will be an IMP network task that does the mapping. Clearly, at the very least, the sending task needs to know when that other task has finished with the data in the shared memory so that it can know when it is safe to delete the section or to modify the data in it. IMP will make sure it gets that information, and also allows it to request more detailed reports on the status of the transfer.

The mechanism, as might be expected, is through IMP system messages. The sending program, once it has called `ImpSend Bulk()`, should check any IMP system messages sent to it. It should expect to get at least one message of a type categorised by `ImpSystemMessage()` as IMP_SYS_BULK_TRANSFERRED. Such a message has a number of fields explained in more detail later, but the important ones give the total number of bytes in the transfer, the number transferred so far, and whether or not the shared memory section used has been released by the other task that has mapped it.

If the target task rejects the bulk data, or is unable to handle it, the sending task will get just one IMP_SYS_BULK_TRANSFERRED message, and it will have a non-zero value in its `Status` field.

When `ImpSendBulk()` is called, the calling routine is expected to set the `NotifyBytes` field in the 'message information' structure used to describe the message. This can be used to indicate how often the sending task would like to be notified of the progress of the transfer. The system will aim to notify the sender every time that many bytes are sent. No matter what value of `NotifyBytes` is specified, the sending task should expect to get at least one reporting message — either the final one that indicates that all the data has been transferred and that the shared memory section has been released, or one that indicates that an error has occurred with the transfer.

It is also possible to associate additional data with the bulk data. This associated data is sent to the target task at the same time as the initial message establishing the bulk data transfer, but is an ordinary byte array such as might be sent in any IMP message. The intention is that many applications will want to send something describing the bulk data they are sending, and will want to know that this additional information is tightly coupled to the bulk data, so there is no question of the target task getting confused as to which description goes with which bulk data.

This sending sequence is just the same whether the target task is on a local machine or on a remote one, and can be summarised as:

1) Make sure you have a connection to the target task that is a two- way connection that can handle messages of size IMP_BULK_MESSAGE_LENGTH.

2) Create the shared memory section and use `ImpDefineShared()` to fill out a structure that can be used to describe it to IMP. (Or you can use `ImpDefineShared()` to create a section of a specified type.)

3) Make sure the required data is in the shared memory section.

4) Call `ImpSendBulk()` to initiate the transfer of the data. If you want to be notified of the progress of the transfer, set `NotifyBytes` to a suitable value. At this point you can also set the `AssociatedAddress` and `AssociatedBytes` fields to specify any associated data.

5) Wait for messages of type IMP_SYS_BULK_TRANSFERRED. These will indicate how much of the data has been transferred so far. Eventually, you will get a message that has the `Released` flag set, and this is an indication that it is now OK to modify the data in the shared memory section or even to delete the section.

6) Delete the shared memory section, unless you want to use it again. You can delete it directly, or through a call to `ImpReleaseShared()`. If it was created for you by `ImpDefineShared` you must release it through `ImpReleaseShared()`.

There is no limitation imposed by IMP on the number of concurrent bulk data transfers that a task can initiate, either to different target tasks or even to the same target task. Equally, a task can be sending and receiving bulk data at the same time. If concurrent transfers are used, the sending and receiving tasks have to be careful to make sure they can tie the IMP_SYS_BULK_TRANSFERRED messages they get to calls they have made to `ImpSend-Bulk()` or `ImpReadBulk()`, or they risk trying to use or detete data before it has been transferred. To do this, they need to make use of the various Tag and Reference numbers that can be set when `ImpSendBulk()` and `ImpReadBulk()` are called, and which are included in the `IMP_BulkReport` structure associated with each IMP_SYS_BULK_TRANSFERRED message.

12.2 Receiving bulk data

The sequence that has to be followed by a task that is coded to receive bulk data is fairly simple. The first a target task knows about a bulk data transfer is when it receives an IMP system message that `ImpSystemMessage()` classes as being of type IMP_SYS_BULK_DATA or IMP_SYS_BULK_WAITING. The difference is that in the first case (the IMP_SYS_BULK_-DATA case) the bulk data is already available in a shared memory section on the local machine, and all the target task has to do is attach itself to that section and make use of the data. In the second case (the IMP_SYS_BULK_WAITING case) the IMP system has bulk data that it wants to send to the target task, but it needs the target task to create a shared memory section to receive the data. The first case is obviously the simplest.

If the sending task and the target task are on the same machine, the data must already exist in shared memory for the sending task to even consider sending it via `ImpSendBulk()`, so the target task will always be notified through an IMP_SYS_BULK_DATA message. If the two tasks are on different machines, the IMP system normally will send the target an IMP_SYS_BULK_WAITING message. This requires that the target set up a suitable shared memory section to receive the data, but it also allows the target to monitor the progress of the data transfer into that section and, of course, to control exactly how the shared memory used is defined.

Some target tasks will find that they want to do exactly that — exercise control over the details of the shared memory section used. Other target tasks will not care and will simply find the whole business of handling IMP_SYS_BULK_WAITING messages unnecessarily complicated. These tasks have the option of registering with the IMP_FORCE_READY flag set (in their original call to `ImpRegister()`). This will force the IMP system to handle any IMP_SYS_BULK_WAITING messages within the networking tasks, and has the effect that such a task will only ever see IMP_SYS_ BULK_DATA messages when bulk data is sent to it. (The task is not sent the IMP_SYS_BULK_DATA message until all the data is transferred to its local machine, into a temporary mapped section created by the IMP networking tasks, which it can then access in the normal way it would handle bulk data sent by a local task.) It is also possible for a sending task to set the IMP_FORCE_READY flag in any specific call to `ImpSendBulk()`. If it does so, that particular bulk data transfer will be handled as if the target task had registered with the IMP_FORCE_READY flag set.

12.2.1 Responding to an IMP_SYS_BULK_DATA message

A target task receiving an IMP_SYS_BULK_DATA system message is being told that there is bulk data already available for it in a shared memory section on the remote machine. The message holds enough information for the target task to access this section and the data it holds. The receiving task can call `ImpHandleBulk()`, passing it the message information structure it got from the `ImpRead()` or `ImpReadPtr()` call that read the original message. `ImpHandleBulk()` will get the necessary information out of the message, will map the shared memory section and will return the address at which the section was mapped, the size of the section in bytes, and an IMP shared memory description structure describing the section.

The target task now has access to the bulk data. Whether or not it is allowed to to modify that data is a matter for it and the sending task to agree on. IMP maps the data in a manner that will allow it to be modified, but provides no specific interface to control write access. IMP makes it easy enough for the two processes to send messages to each other, if you want to code some private convention controlling this sort of thing. When the target task has finished with the data, it should call `ImpReleaseShared()`, passing it the shared memory description structure returned by `ImpHandleBulk()`. The target task should not normally specify that the section be deleted — this is usually up to the sender. Then it needs to let the sending task know that it has released the memory section, which it does through a call to `ImpBulkReport()`.

`ImpBulkReport()` needs to be passed an IMP bulk report structure which contains enough information to identify the bulk transfer in question to the sending task. When `ImpSystemMessage()` processes the original IMP_SYS_BULK_DATA message it sets up a `BulkReport` structure in its `SysInfo` argument that can be used for this purpose. The target task merely has to keep a copy of this structure and set the `TransferredBytes` field to equal the `TotalBytes` field (indicating that all the data has been read) and to set the `Released` flag to indicate that the shared memory section has been released. Each call to `ImpBulkReport()` results in an IMP_SYS_BULK_TRANSFERRED message being sent to the original sending task. It is quite reasonable for the target task to make two calls to `ImpBulkReport()`, one to indicate that all the bulk data has been received (ie with `TransferredBytes` set equal to `TotalBytes`, but with `Released` not set) and some time later a second call (with `Released` now set) to indicate that the shared memory section has been released.

So the sequence that should be triggered by an IMP_SYS_BULK_DATA message is pretty simple, and is as follows:

1. Save the contents of `SysInfo.BulkReport` as returned by `ImpSystemMessage()`, since this is the easiest way to identify the bulk data.
2. Call `ImpHandleBulk()` to map the shared memory section into the address space of the target process.
3. Optionally, at this point call `ImpBulkReport()` to indicate that the data has been received, but not yet released.
4. Make use of the data.
5. Call `ImpReleaseShared()` to unmap the shared memory section.
6. Call `ImpBulkReport()` to indicate that the memory section has been released (ie. with the `Released` field set true).

12.2.2 Responding to an IMP_SYS_BULK_WAITING message

A target task receiving an IMP_SYS_BULK_WAITING message is being told that another task (as it happens, always a remote task) has bulk data that it wants to send to the target task. The target task in this case has the responsibility of providing a shared memory section into which this bulk data can be transferred. The target task may already have a suitable section that it can use or it may have to create one. In any case, it needs to call `ImpDefineShared()` to initialise an IMP shared memory description structure with details of the mapped section. If it simplifies things, `ImpDefineShared()` can create the mapped section from the description provided.

(Remember, if this is starting to sound too complicated, a task may always opt to set the IMP_-FORCE_READY flag in its original call to `ImpRegister()`. This guarantees it will never see an IMP_SYS_BULK_WAITING message — all bulk data transfers will be handled as needed by the IMP system so that all it ever sees are IMP_SYS_BULK_DATA messages.)

At this point, the target task may not wish, or may not be able, to receive the bulk data. The call to `ImpDefineShared()` may have failed — there may not be enough disk space left to create a suitable mapped file, for example — or the task may have its own reasons for rejecting the transfer. If this is the case, it should call `ImpReadBulk()` as described in the next section, but with a non-zero value for the `LocalStatus` parameter. If `ImpDefineShared()` has indeed failed, the bad status value it returned should be used. Otherwise, if the task wants to reject the transfer for its own reasons, it should use any non-zero value — preferably one that the sending task will recognise through mutual agreement.

Once a suitable shared memory section exists, the target task should call `ImpReadBulk()`, passing it the shared memory description structure set up by `ImpDefineShared()`. This will initiate the transfer of the bulk data into that memory section. The question for the target task now is one of knowing when the data has been written into the memory section so that it can access it. In the same way that the sending task is notified of the progress of the transfer through IMP_SYS_BULK_TRANSFERRED system messages, the target task will also receive such messages, in this case indicating the state of the transfer into its shared memory.

The target task can specify that it wants to be updated as to the state of the transfer as it progresses, rather than just at the end, by setting the NotifyBytes field of the `MsgInfo` argument it passes to `ImpReadBulk()` to a non-zero value. In any case, it will eventually receive an IMP_SYS_BULK_TRANSFERRED message that has the `TransferredBytes` field equal to the `TotalBytes` field, which will indicate that all the data has been transferred. Usually, the same message will also have the `Released` field set to indicate that the task writing into the shared memory has released the memory. Once it gets the system message indicating that the memory has been released by the writing task, the target task is free to unmap the shared memory section, should it wish to, either directly or through a call to `ImpReleaseShared()`. Normally, the target task will set the `Delete` flag in this call.

Although it is not required by the IMP system, a target task may choose to call `ImpBulk-Report()` to notify the original sending task that it has received the data. The sending task will have been notified by the IMP system that the data has been sent over the network, and this should normally be enough, but if a sending program really wants to be told by the final target task itself that the data has been delivered then `ImpBulkReport()` can be used for this task. Or, of course, an ordinary IMP message with some agreed format can be sent between the two tasks. The point is that although it isn't really IMP's job to provide complete communications conventions between tasks (this is normally left to a higher level layer), `Imp-BulkReport()` can do this particular job and there's no reason not to use it. The sending task will get an IMP_SYS_BULK_TRANSFERRED message that has the `FromTarget` field set to indicate that this message is indeed from the target task and not some IMP intermediary.

So, the sequence that should be triggered by an IMP_SYS_BULK_WAITING message is also pretty simple, and is as follows:

1. Create a suitable shared memory section and use `ImpDefineShared()` to fill out a structure that can be used to describe it to IMP. (Or you can use `ImpDefineShared()` to create a section of a specified type.)
2. Call `ImpReadBulk()` to initiate the transfer of the data. If you want to be notified of the progress of the transfer, set `NotifyBytes` to a suitable value. If stage 1) failed, pass a non-zero status (for, example, the status value returned by `ImpDefineShared()` as the value of `LocalStatus`, and ignore the remaining stages.
3. Wait for messages of type IMP_SYS_BULK_TRANSFERRED. These will indicate how much of the data has been transferred so far. Eventually, you will get a message that has the `Released` flag set and that indicates that all the data has been transferred.
4. Make use of the data in the shared memory section. (If you were notified of data arrival during the course of the transfer, you can make use of the data that has arrived as soon as you get this notification.)
5. Delete the shared memory section, unless you want to use it again. You can delete it directly, or through a call to `ImpReleaseShared()`.
6. Optionally, call `ImpBulkReport()` to let the sending task know the state of play at the target task end. (If you do this, you should have kept a copy of the `SysInfo.Bulk-Report` structure returned by the original call to `ImpSystemMessage()` to pass to `Imp BulkReport()`.)

12.3 System-dependent aspects of calling `ImpDefineShared()`

The call to `ImpDefineShared` allows a shared memory section to be defined through a combination of a numeric `Type` code, a character string (`Name`), an integer (`Key`) and an address (`Address`). (Note that the `Address` argument is the address of a void pointer, since this can be an input or an output argument, depending on the value of the `Create` parameter.) These can be used to specify a shared memory section of any of the types supported by the systems on which IMP has been implemented so far, and even perverse new systems can probably be handled by contrived usees of the `Name` string if necessary.

It should be noted that if you are not concerned about the details of the shared memory section created by `ImpDefineShared`, you can simply set the `Type` code to IMP_SHARE_CREATE. In this case, the input values of all of the `Name`, `Key` and `Address` arguments are ignored, as is `Create`, which is assumed to be true. This will create a shared memory section of the required size on any system, and provides a system-independent way of creating a shared memory section.

The ways shared memory sections are handled on the systems supported at present are explained in the descriptions given below, which have been extracted from the `ImpDefineShared` header comments. The codes used for `Type` are defined in the imp.h file.

VMS The mapped section is determined entirely by the `Name` field, which should be the name of a global page section. `Type` should be passed as IMP_SHARE_GBLSC. `Key` and `Address` are ignored. Note that `Name` can be almost anything. If `Type` is zero, IMP_SHARE_GBLSC is assumed. `Type` can always be set to IMP_SHARE_ CREATE, if a system-independent call is required and no control over the shared memory section is needed.

VxWorks The mapped section is just a section of memory, starting at a specified address. `Type` should be IMP_SHARE_GLOBAL. `Name` and `Key` are ignored. If `Create` is specified, `Address` is ignored, and a suitably sized area of memory is allocated. If `Create` is passed as false, then `Address` should contain the address of the memory section in question. If `Type` is zero, IMP_SHARE_GLOBAL is assumed. `Type` can always be set to IMP_SHARE_-CREATE, if a system-independent call is required and no control over the shared memory section is needed.

UNIX The mapped section can be created as System V shared memory, in which case `Type` should be IMP_SHARE_SHMEM, or as a file accessed through `mmap()`, in which case `Type` should be IMP_SHARE_MMAP. If `Type` is IMP_SHARE_SHMEM, `Key` specifies the identifier for the shared memory and `Name` is ignored. If `Type` is IMP_SHARE_MMAP, `Name` should be the full name of the file, and `Key` is ignored. `Address` is ignored in both cases. If `Type` is zero, IMP_SHARE_SHMEM is assumed if the system supports the use of memory mapped files, and IMP_SHARE_SHMEM is assumed if it does not. `Type` can always be set to IMP_SHARE_CREATE, if a system-independent call is required and no control over the shared memory section is needed.

Note that the design of the IMP bulk transfer routines never requires that a program understand the contents of an `IMP_SharedMemInfo` structure as returned by `ImpDefineShared()`. A program is never required to create or delete a structure as specified in such a structure. The structure contents should be regarded as an internal matter for IMP. The requirement that such a structure be capable of transmission over a network, for example, means that it cannot include addresses in a straightforward way, since address length can differ from machine to machine. This sort of consideration makes the actual structure contents awkward to work with and best hidden from the use by the IMP routines. (The question of why such a structure needs to be capable of network transmission is one for the IMP internals manual rather than this document.)

13 Structures used by IMP

Programs using IMP need to include the file “imp.h”. This defines a number of structures and types that the program will have to make use of, since they are used to pass information to the various IMP routines. This section describes those that programs written to use the IMP routines will need to use:

13.1 The IMP_ID Process Identifier

When a process registers itself with IMP through a call to `ImpRegister()` it is returned an identifier for the process. This is something of type `IMP_ID`. Once the user has been given a value for this, it has to be supplied in almost all subsequent calls to IMP routines made by the process. The identifier itself is something that the user code should not change in any way, and it should resist the temptation to make use of it other than to pass its value to the IMP routines.

(You might as well know that the `IMP_ID` for a process is actually a pointer to a structure in which IMP retains all the information connected with that process. This has to be supplied in each call to IMP since there are systems — such as VxWorks — where there is only one version of the IMP code used by all the various processes and where there are undesirable overheads associated with having the IMP system locate this process-specific information for itself; you can’t just stick it in a single static variable.)

13.2 The IMP_MsgInfo Message Information structure

The main structure that has to be manipulated by routines calling the IMP routines is an IMP ‘Message Information’ structure, `IMPZ_MsgInfo`. A structure of this type contains the following fields (and perhaps some others which are used internally by the IMP routines):

`.Address` (`void *`) This is used as the address of the message. For routines that send messages, this has to be set by the caller to the address of the start of the message to be sent. For routines that read messages into buffers, this has to be set by the caller to the address of the start of the buffer that is to receive the message. Routines such as `ImpReadPtr()` which read a message and return a pointer to it, return the pointer in this field.

`.AssociatedAddress` (`void *`) If bulk data is received, and if additional data was associated with the bulk data being sent, `AssociatedAddress` will indicate the address at which that associated data may be found.

`.AssociatedBytes` (`int`)If bulk data is received, and if additional data was associated with the bulk data being sent, `AssociatedBytes` will be set to the length in bytes of that associated data. If no associated data was sent with the bulk data, `AssociatedBytes` will be set to zero.

`.BufferLength` (`int`) When a message buffer is supplied to an IMP routine for a message (with its address in the `.Address` field) this field should be set to indicate the length of the buffer in bytes. This ensures that a reading routine will not overwrite the end of the buffer. Note that this is not the same as the `.MessageLength` field. `.BufferLength` is used only to tell a reading routine the length of the buffer it is to read into.

`.DeltaTime` (`IMP_DeltaTime`) If a time interval has to be supplied to an IMP routine, such as the timeout value in a call to `ImpRead()`, or the delay value in a call to `ImpQueueRem-inder()`, this field is used to supply the time interval in question. The format of this field varies from implementation to implementation, so this field should always be set using a call to `ImpDeltaTime()`.

`.MessageLength` (`int`) This field is used to hold the length in bytes of a message. For a call to a sending routine such as `ImpSend()` the caller should set this field to the length of the message to be sent. A reading routine such as `ImpRead()` will set this field to the actual length of the message read.

`.ReadFlag` (`int`) The reading routines such as `ImpRead()` set this field to indicate whether or not a message was actually read. This will normally be set to a true (non-zero) value, but will be set to zero to indicate a timeout, or a read call that specified 'nowait' when there was no message to be read. A calling program will know whether either of these cases are possible, and should test this flag if they are.

`.Tag` (`IMPZ_INT4`) This field allows the sending routine to tag the message with an identifier of its own choosing. When a message is sent, the calling routine should set this field to the required tag value before calling the sending routine, eg `ImpSend()`. When a message is read, a routine such as `ImpRead()` will return the type value in this field. Also, if a message cannot be delivered, the message returned to the sending task will include the tag of the message in question to allow it to be identified. Note that the tag is transmitted with the message in a four byte long field, so it is defined using a system dependent type (`IMPZ_INT4`) which is defined in the IMP include files as a four byte integer.

`.Type` (`IMPZ_INT4`) This defines the type of the message. Negative type values are reserved by IMP for its own internal use. When a message is sent, the calling routine should set this field to the required type before calling the sending routine, eg `ImpSend()`. When a message is read, a routine such as `ImpRead()` will return the type value in this field. Messages with negative values for this field should be processed by `ImpSystemMessage()` after they have been read. Note that the value is transmitted with the message in a four byte long field, hence the type used (see the description of the Tag field.)

13.3 The IMP_TaskID Task Identification structure

The 'Task Identification' structure, `IMP_TaskID` is used to identify tasks on the system, and contains enough information to identify a task uniquely — at present it contains a process ID and a machine ID. Programs using IMP should not normally need to access this information directly. Each `IMP_MsgInfo` Message information structure contains a task identification structure giving the identification of the sending task, and the address of this structure can be obtained through a call to `ImpFromTask()`. Normally, a task identification will be obtained in this way and this can then be passed to a routine like `ImpConnect()` or `ImpConnection()` to get a connection to that sending task. A task identification structure is returned by `ImpSystemMessage()` to identify the task referred to in the message, which is not always the same as the task sending the message.

13.4 The IMP_SysInfo System Message Information structure

This structure is used by the routine `ImpSystemMess age` to return information about a system message that it has handled. This routine sets the `.SysMsgType` field to a code describing the type of system message it has just handled, and then sets the rest of the fields as appropriate to the message type. A full description of the possible values for this message type and the exact meaning of the other fields set for each message type can be found in the detailed description of the `ImpSystem Message` routine. The fields contained in this structure are:

`.BulkReport` (`IMP_BulkReportInfo`) This is a structure containing all the information pertaining to the progress of a bulk data transfer. It contains a number of tag values and reference numbers to identify the transfer in question, and has a `TransferredBytes` and a `TotalBytes` field that show how much data has been transferred so far and how much data there was to be transferred. It also contains a `Status` field that will be non-zero if there has been an error in the transmission of the bulk data.

`.InputNumber` (`int`) In all cases where an input connection number is reported by `ImpSystem-Message` this is the field set to that connection number.

`.SysErrText` (`char[]`) When a system-dependent code is set in the `.SysSysStat` field, this string contains a translation of the status, which is generally more informative than a system code from an unknown machine.

`.SysMsgType` (`int`) The code describing the type of system message just handled.

`.SysMsgStat` (`IMP_Status`) A status code, usually describing the status of an operation which the message is reporting as having completed. This is always an IMP__ code, with zero indicating success.

`.SysSysStat` (`long`) When a task loaded in response to a `ImpRunTask()` call terminates or fails to load, the task that requested its load is sent a system message including the system- dependent failure code or exit status and this is returned in this field. System codes generated in heterogeneous systems are not generally helpful, and exit codes are not always informative. A success code is always set to zero, even where the system uses some other convention to indicate 'success'. returned in some cases where an error is detected on a remote machine. This is not often much use, since the

`.TargetTask` (`char[]`) Of interest mainly to 'Translator' tasks, which have to be able to distinguish between the different named tasks on whose behalf they act. When a new connection is set up, this indicates the name of the task at which it is directed.

`.TaskName` (`char[]`) The name of the task in question, where a message refers to a task identified by name.

`.TaskID` (`IMP_TaskID`) Used to identify the task, if any, identified in the message.

`.Tag` (`int`) When the message refers to another message, the Tag field gives the tag value associated with that message.

13.5 The IMP_BulkReport structure

The description of how the bulk data routines operate should have shown that the whole bulk transfer system in IMP amounts to little more than having the tasks in question call the IMP bulk transfer routines and react to IMP_SYS_BULK_DATA, IMP_SYS_BULK_WAITING and IMP_SYS_BULK_TRANSFERRED system messages. Reacting properly to these is to some extent a case of understanding the contents of an `IMP_BulkReport` structure. The `IMP_SysInfo` structure that a program has to pass to `ImpSystemMessage()` to get information back from that routine contains an `IMP_BulkReport` structure called `.BulkReport`. And it is a structure of type `IMP_BulkReport` that has to be passed to the `ImpBulkReport()` routine.

In the case of the IMP_SYS_BULK_DATA message, a call to `ImpHandleBulk()` extracts almost all the necessary information from the message. It maps the shared memory and returns an `IMP_BulkDataInfo` structure describing it together with the size of the bulk data and the address at which it has been mapped. The only other information that a target task needs is contained in the `IMP_BulkReport` structure that is returned by `ImpSystemMessage()` as part of its `SysInfo` argument.

In the case of the IMP_SYS_BULK_WAITING message, almost all the information needed is in that same `IMP_BulkReport` structure. Its `TotalBytes` field gives the size of the bulk data in bytes, which is what the target task needs to know in order to create a suitable shared memory section for the data.

In the case of the IMP_SYS_BULK_TRANSFERRED message, all the information in the message is returned by `ImpSystemMessage()` in that same `IMP_BulkReport` Structure.

So it is worth explaining in detail what is in an `IMP_SysBulkReport` structure. It contains the following fields:

`SenderTag` is the Tag value specified by the sending task in the original call it made to `ImpSendBulk()`.

`SenderRef` is the RefNumber value specified by the sending task in the original call it made to `ImpSendBulk()`.

`TargetTag` is the Tag value specified by the target task in the original call it made to `ImpReadBulk()`.

`TargetRef` is the RefNumber value specified by the target task in the original call it made to `ImpReadBulk()`.

`TransferredBytes` is the number of bytes of bulk data actually transferred so far.

`TotalBytes` is the total number of bytes to be transferred — this is taken from the size of the shared memory section as passed to `ImpSendBulk()`.

`Status` is a status value. If non-zero, this is an indication that there has been an error in the transfer. If a target task calls `Imp ReadBulk()` with a non-zero value of `LocalStatus`, the originating task will get a system message classed as IMP_SYS_BULK_TRANSFERRED which wil have the `Status` field set to this non-zero value.

`Released` is a flag that is set to indicate that a task has released — unmapped — the shared memory section in question. For a remote transfer the task in question will be an IMP network task, for a local transfer it will be the actual target task.

`FromTarget` is a flag indicating that the message originates with the final target task for the data, rather than from the IMP networking system.

`RefersToASend` is a flag indicating that the message is being sent to a sending task rather than to a target task. This may be necessary if a task is both sending and receiving bulk data and needs to know if it should look at the sender or target tags and reference numbers to identify the transfer in question.

`TargetNotifyBytes` is the `NotifyBytes` value specified by a target task in a call to ImpRead-Bulk().

`SenderNotifyBytes` is the `NotifyBytes` value specified by the sending task in a call to Imp-SendBulk().

14 ‘Translator’ tasks

The concept of the ‘Translator’ task was introduced in the first place to enable tasks using the AAO’s DRAMA system (a real-time distributed data acquisition system that uses IMP for inter-task communication) to communicate with an older set of data acquisition tasks that were written using the ADAM system. ADAM is a similar system to DRAMA, but uses its own message system for inter-task communication. The intention was to allow an intermediate task (a ‘translator’ task) to be linked with both message systems and to act as an interpreter between the two systems. Knowing this may make it easier to see how a translator task is intended to be used.

The IMP side of the operation of a ‘Translator’ task is as follows:

1. The task registers with IMP by calling `ImpRegister()` with the `IMP_TRANSLATOR` flag set.

2. If it is already aware of the names of some or all of the tasks that it is going to translate for, it can register them immediately by calling `ImpProxyRegister()` on their behalf. Once a task has been registered in this way, it appears to the system just like any other task, but when a task tries to make a connection with it, the connection is actually made with the translator task.

3. When a task calls `ImpNetLocate()` for a task that is not registered, the system will look to see if there is a translator task registered. If this is the case, the translator task will be sent a message that `ImpSystemMessage()` will classify as an `IMP_SYS_ENQ_HANDLE` message. This message includes the name of the unregistered task in question. The translator task must respond to this message by a) working out if it does in fact handle this task (this may involve some enquiry operations with the non-IMP part of the system, but that is not any business of IMP), b) calling `ImpProxyRegister()` to register the task if it does handle it, and then c) calling `ImpHandled()` with the ‘`Handled`’ argument set false or true depending on whether the task is handled or not.

4. As a result of a task being registered by a translator task through a call to `ImpProxy-Register()`, other tasks can attempt to make connections with it. Any connection with a translator task, whether one-way or two-way, will result in a message that `ImpSystemMess age()` will classify as an `IMP_SYS_CONNECT` message being sent to the translator task. The translator task must then call `ImpAccept()` to accept the connection (it can reject the connection, also using the call to `ImpAccept()`). Once the connection is accepted, it may start to get messages on that connection. At this point, it should make a note of the input connection number and target task name returned to it in the structure of type `IMP_SysInfo` by `ImpSystemMessage()`, so that it can tell for which task any subsequent messages sent on that connection are intended.

5. Normally, when a call to `ImpRead()` or `ImpReadPtr()` returns a message, the calling task does not know or care about the input connection on which the message arrived. A translator task, however, needs to know this in order to identify the target task for the message. (The message header has an `IMP_TaskID` structure that identifies the target task by Pid, but since the tasks handled by a translator task may not even have separate Pids this cannot be used to distinguish them and so this will always turn out to specify the translator task itself.) So, for any message it gets, the translator task should call `ImpInputNumber()` to get the input connection number, should use this to identify the target task name associated with this connection, and should handle the message accordingly.

6. A translator task should take particular notice of system messages classified by `Imp-SystemMessage()` as being of type `IMP_SYS_CONN_CLOSE` since these indicate the closing of an input connection and mean that the association between this input connection and the corresponding target task name should be broken. The '`InputNumber`' field in the `IMP_SysInfo` structure returned by `ImpSystemMessage()` gives the input connection number in question.

7. If a translator task wants to indicate that one of its handled tasks has closed down, it should call `ImpProxyDetach()` for that task. The effect is the same as if the actual task had called `ImpDetach()`.

8. Eventually, the translator task can close itself down by calling `ImpDetach()`.

These steps are illustrated by the example program '`ttran`' which appears in this document. This registers under a specified name, but procedes to respond positively to any 'do you handle this task?' enquiries, no matter what name they specify, and to accept connections on behalf of any of these tasks. It then echoes messages sent to any of these tasks. If a 'QUIT' message is sent for any of the tasks it handles, it acts as if that task has exited. Once it has received 'QUIT' messages for all the tasks it handles, the task itself exits. This is hardly a realistic example, but it demonstrates most of the functions of a translator task.

15 Network startup

Version 1.0 of IMP introduced the concept of the 'network startup file'. This is a text file read by the networking tasks on startup which can be used to control certain aspects of their operation. At present, only a very limited number of options (four, in fact) are supported, but now that the mechanism exists others can be added quite easily.

The startup file has the name '`IMP_Startup.nodename`' where '`nodename`' is the internet name of the machine on which the network tasks are running. Having the nodename present means that the startup files for a number of different machines can be held in the same, shared, directory. Note that the string used for '`nodename`' should match that returned by `ImpNodeName()` and may not always be what you expect, since it depends on how the networking has been set up. (Our VAX 4000, for example, which I always think of as 'AAO40A', in fact returns 'AAOEPP3' as its nodename.)

The programs look in a number of directories for this startup file. First, they look in any directory with the symbolic name '`IMP_STARTUP`'. (Under UNIX, this is an environment variable, under VMS this is a logical name.) If no startup file is found there, they look in the current default directory, and if no startup file is found there, they look in the directory from which the 'Transmitter' task is being run, if this can be determined.

Note that this means, amongst other things, that the port used by IMP can be controlled by having the symbolic name '`IMP_STARTUP`' defined in such a way that it causes IMP to use a startup file containing a suitable '`Use port`' command.

Blank lines in a startup file are ignored. If a line contains a '`#`' or a '`!`' character, then this is taken as introducing a comment and the character and any following it on the line are ignored. The case of all characters is ignored. The options that are recognised are the following:

```
Log startup sequence
Use port {n} [for {machine}]
Connect at startup to {machine}
Check connections to {machine} every {time spec}
Pulse connections to {machine} every {time spec}
Note alias {machine} for {machine}
```

In the above list, something included in square brackets is optional. Something included in {curly braces} is required but needs to be replaced in the actual command line with a suitable value. {`n`} indicates an integer port number. {`machine`} indicates a remote machine, specified using a recognised internet address ('`aaossz`', or '`aaossz.aao.gov.au`' or '`123.456.789.123`', for example). What constitutes a valid time specification is discussed in more detail below, but strings like '`20 seconds`' or '`5 minutes`' are accepted.

If the option specified is

```
Log startup sequence
```

then the startup file and any actions performed as a result of reading it are logged. This line would normally be at the start of the file, if it is included at all.

If the option specified is

```
Use port {n} [for {machine}]
```

for example

```
      Use port 23456 for aaossz
```

this specifies that the IMP system will use the specified port number when making TCP/IP connections to remote machines. If a machine is specified explicitly, then the port number is used just for that machine. If no machine is specified, then the number specified is used for all machines including the local machine. You can have this command as often as you like in the startup file, so you can, for example, specify a general default port number to use and then specify different port numbers on a machine by machine basis. By default, IMP uses a port number generated from the name of the current user — this was convenient in its initial testing phases, but more control is needed now. A more sophisticated scheme will eventually be introduced, but for the moment this should provide enough control for most purposes.

If the option specified is

```
     Note alias {machine} for {machine}
```

for example,

```
     Note alias aaossz2 for aaossz
```

this indicates that the machine normally referred to as 'aaossz' has more than one IP address that it can respond to. This is provided for the case where machines have more than one network connection. The example line above indicates that 'aaossz2' is also a recognised internet name for the machine normally known as 'aaossz', but that 'aaossz2' translates to a different internet address to that obtained by looking up 'aaossz'. This implies that accessing the machine as 'aaossz2' will use a different network connection to that used when the machine is accessed as 'aaossz'. This can, for example, be used to access a private connection between two machines intended to be dedicated for a particular purpose.

IMP needs to be explicitly informed about such multi-addressed machines, For example, when it is told that 'aaossz' has gone down it needs to be able to deduce that all its connections will 'aaossz2' will also now be broken. The term 'alias' here might be a misnomer, but the facility it provides is important. Note that this is for use only with multi- addressed machines; it is not intended for the common case where a number of different internet names all translate to the same numeric IP address.

The three options:

```
      Connect at startup to {machine}
      Check connections to {machine} every {time spec}
      Pulse connections to {machine} every {time spec}
```

are all mainly connected with trying to provide robust detection of machines that disconnect from the network. The following discussion is rather long-winded, but may be worth following. If you have machines where there is a risk that they will disconnect without warning (and this is particularly the case with VxWorks systems), it is worth using one of these options to pick up such events. The best one to use is the 'Pulse connections' option. The others are rather inferior, although they put slightly more load on the network.

If the option specified is

```
Connect at startup to {machine}
```

for example,

```
Connect at startup to AAO40A
```

then the transmitter task will attempt to make a connection to that machine as it starts up. Note that the connection is attempted as soon as the line is read from the startup line, so if you want it to use a port other than the default you should have the relevant '`Use port`' line *before* the '`Connect at startup`' line.

There may not seem much point in forcing a connection to a specified machine. After all, any connection will be made as needed when requested by a task on the local machine. There is a small advantage in that the overhead for the connection will be taken at startup rather than when the connection is requested by a client task, but the real reason this feature was introduced was an attempt to make it easier to pick up crashed machines.

Most machines don't crash very often, but when they do, the Internet protocols are such that machines they're connected to often don't get notified and so miss the fact that the remote machine has gone down. This is not usually an issue for UNIX or VMS machines, but VxWorks is a much more fragile operating system, and VxWorks machines, particularly when real-time systems are being developed, can crash quite often. If a VxWorks machine is set up to initiate a connection to a specific remote machine on startup, then when the remote machine receives this connection request (which is identified as a 'startup' request) it knows quite definitely that the machine in question is restarting and any tasks it thinks it was in contact with on that machine are defunct (and have been so for a period of time). This makes it much easier for IMP to close down any connections with the now rebooting machine and to clean up things in general.

Unfortunately, this scheme doesn't work as reliably as was originally hoped. Under some circumstances an attempt by a newly-restarted VxWorks machine to connect to a machine to which it had previously been connected would fail. After some investigation, it seemed that the only reliable way to pick up the crashing and restarting of a VxWorks system was to have an outstanding read on the VxWorks system. (People who have used rlogin, for example, to log on to a VxWorks system will know that it is always in a state of reading from the VxWorks system and if the VxWorks system is crashed the rlogin shell will pick this up reliably — but only once the VxWorks system reboots.) IMP is not in a position to keep such a blocking read current on a VxWorks system, but it can get the same effect by occasionally testing a connection to a VxWorks system by issuing a read to it. Hence the introduction of the startup option:

```
Check connections to {machine} every {time spec}
```

for example

```
Check connections to aaovme2 every 10 seconds
```

This causes the IMP Receiver task to test any connections it has to the specified machine at the specified interval. The interval is specified by an integer followed by a word that should begin with either '`sec`', '`min`' or '`ho`', interpreted as seconds, minutes and hours respectively. This seems to pick up VxWorks machines restarting after a crash and to do so reliably. The interval needs to be small enough that a read will be issued between a machine crashing and its attempting to make a new connection to the testing machine.

However, even this doesn't address the problem of a system that merely disconnects or which crashes without being restarted. You can get this effect, for example, by physically disconnecting the network cable from a VxWorks machine. The only reliable way to pick this up is to force a handshake on a regular basis between the machines in question, and this is done using the option:

```
   Pulse connections to {machine} every {time spec}
```

for example

```
   Pulse connections to aaovme2 every 10 seconds
```

This runs a utility program called 'imppulse' on the local machine which will send the 'Receiver' task on the remote machine a test message at the time interval specified. Normally, the remote machine will echo this message back. If it has not done so before the time interval expires and 'imppulse' is ready to send another test message, 'imppulse' will deduce that the machine has disconnected and will signal this to the rest of the IMP system. This option seems finally to provide a reliable means of picking up disconnecting VxWorks systems. The time delay needs to be more than the longest anticipated response time from the remote machine, but short enough to pick up the problem satisfactorily quickly. For local machines, intervals such as 10 seconds are usually fine.

16 When IMP doesn't work

The most common problem with IMP programs comes from needing to specify enough memory, both when a program first registers (in the call to `ImpRegister()`) and when the size for a particular connection is specified (in the call to `ImpConnect()`. If you get any error messages along the lines of 'insufficient memory to allocate buffer' or 'not enough space in ring buffer' then you should look first at the amount of memory specified in these calls. Put it up a lot and see if you still have the problem! (You can often need more than you expect; for example, the '`atest`', '`ttest`' test programs only send one message back and forth between them, but because they use reading by pointer it is possible for `ttest` to read a new message, echo it to `atest` and have received the echoed response *before* calling `ImpReadEnd` to clear the original message from its buffer. This means it can have two copies of the message in its buffers at once when a first guess would suggest that it only needs space for one.)

Note that use of `ImpErrorText()` can make it much easier to produce comprehensible error messages from within IMP programs. There should be calls to `ErsFlush()` at regular intervals, or when an error occurs, in order to make sure that error information buffered by the Ers system is output properly. (If you get messages about ERS buffers overflowing, you are not calling `ErsFlush()` to flush out the buffered messages.)

On UNIX systems, you can get dreadful response from IMP by putting the memory-mapped scratch files on a remote disk mounted through NFS. This will show up particularly for large messages.

If tasks have trouble registering it is often because older versions are still present in the system, or (especially under UNIX), because some of the IPC (Inter process communications) mechanisms used are hanging around in the system. VMS systems don't usually have this problem, but under UNIX it is often necessary to run the '`cleanup`' program supplied with IMP to get rid of the IPC mechanisms and any files previously used for memory mapping. Generally, if IMP gives trouble under UNIX, the first step of all is to run '`cleanup`'.

There is a utility supplied with IMP called '`impdump`'. Running this produces a very detailed snapshot of the IMP internal information for all the tasks on the system. Fighting through all this information to get anything useful from it is not easy, but there are times when this can be used to diagnose IMP problems. The output from '`impdump`' can be a very useful adjunct to a bug report. The '`impdump`' program accepts a number of qualfifers than can be used to restrict its output; the most common option is '`impdump -t taskname`' which restricts the output to details of the task that registered with IMP under the name 'taskname'. The comments at the start of the '`impdump`' code list all the currently accepted options. (Bug reports should be sent to '`ks@aaoepp.aao.gov.au`').

Another IMP utility is '`netmonitor`' which can be used to monitor messages, particularly system messages, and this can be used to diagnose handshaking and similar problems. For details see the comments at the start of the '`netmonitor`' code. A document called 'IMP diagnostics' is available, and this describes in detail the various diagnostic facilities built into IMP and the utility programs that come with IMP for use with these facilities. It is, for example, possible to make an IMP task dump all its recent message traffic to a file and then look at the contents of that file. It is possible to make the network tasks dump a log of all the recent events that they have handled, both socket events and IMP message events.

17 System-dependent considerations

Although IMP is pretty much the same on all systems, there are a few system dependent considerations, and these are listed in this section. Many of these are connected with the use of symbols (VMS logical names, UNIX environment variables, etc) to control aspects of the way IMP works.

17.1 UNIX in general

Under UNIX, a number of environment variables are used for detailed control of IMP, and there are some considerations connected with the use by IMP of the System V inter-process communications mechanisms.

Use of signals In most cases, IMP tries not to interfere with any other programming techniques that might be used in an applications program. One exception is the use of the SIGALRM signal. If a program uses any IMP timing mechanisms, either using `ImpQueueReminder()` or setting a timeout in a call to `ImpRead()` or `ImpReadPtr()`, this will involve IMP's establishing a SIGALRM signal handler. This will almost certainly interfere with any such handler established by the application.

Location of mapped files On UNIX systems that support the '`mmap()`' file-mapping system call, IMP uses this to provide shared memory. This means that any shared memory used by IMP must be mapped onto a file, and this in turn means that the file must reside somewhere. The obvious place, '`/tmp`', is not usually satisfactory since a number of systems don't have enough space here for multi- megabyte files (ours often doesn't). The environment variable '`IMP_SCRATCH`' is used to specify a directory to be used for this purpose. This should be set to the full path name of the directory to be used (with or without a final '`/`'), and IMP will create a directory in it called '`imp_scratch`' in which it will place the temporary files used for file mapping. Normally, these files will be deleted when no longer needed, but in some cases, particularly if tasks crash, some may be left behind. This directory should be purged on occasion if it seems to be filling up with obsolete files. If '`IMP_SCRATCH`' is not defined, IMP will attempt to use the user's home directory (by translating the '`HOME`' environment variable), and if this cannot be translated it will fall back on using the current default directory. Be careful: a really good way to slow IMP down to a crawl when sending large messages is to put the mapped files used for shared memory on a remotely mounted disk! (Note that all the UNIX systems supported by IMP, with the one exception of ULTRIX, support file mapping using '`mmap()`').

Transmitter buffer size The environment variable '`IMP_NET_KBYTES`' can be set to some integer value to specify the number of KBytes of memory to be requested by the Transmitter task when it loads. If large network messages are to be sent this will need to be set, since the default value is only 1024 (that is, 1024 KBytes are allocated). This is really only necessary in the absence of the long-promised mechanism where memory is allocated dynamically for new connections.

Location of network startup files The environment variable '`IMP_STARTUP`' is used to control where the network tasks look for any startup files. This is described in detail in the section on 'Network startup files'.

Location of network tasks The environment variable '`IMP_DIR`' is used to specify the directory where the IMP Master task looks for the network tasks ('Transmitter' and 'Receiver').

If this is not defined, or if the executables for these tasks are not present in the specified directory, the Master task looks in the directory from which it itself was run and then at the current default directory.

Cleaning up The System V inter-process communication (IPC) mechanisms used by IMP, once created, remain in the system until explicitly released by all the processes that have used them. If a process crashes without releasing its IPC mechanisms then they stay in the system, using resources and generally getting in the way. IMP provides a utility routine called '`cleanup`' which removes all the IPC mechanisms that it finds, together with any IMP tasks it finds in the system, that are owned by the current user. Running '`cleanup`' is also the quickest way to close down all the current IMP tasks lock, stock, etc., and is commonly used by lazy people like the author of IMP. If an alias to '`cleanup` is created with the name '`showimp`'this can be used to show the current IMP status (tasks running, IPC resources used, etc.) without actually deleting anything.

IPC key numbers If any use is made of the various System V IPC mechanisms (semaphores, message queues, shared memory sections) directly in code that has to coexist with IMP programs, there is the possibility of clashes with the key numbers used by IMP. If you aren't using calls to '`semget()`', '`shmget()`', '`msgget()`', this isn't going to be something you normally have to worry about. If you are using these routines, you usually have to supply them with a numeric 'key' to uniquely identify the semaphore, message queue or shared memory section you are using. You need to know that IMP usually chooses key values that match the Process Id numbers of the tasks involved. On most UNIX systems, these will always be less than 32768, so choosing values larger than this is usually a good way to avoid conflicts with the numbers used by IMP.

17.2 SUNs in particular

SUN systems usually do not include by default support for the IPC mechanisms used by IMP. If they do, they do not always provide sufficient resources. Unfortunately, this means that to run IMP on SUN systems the kernel often needs to be rebuilt. Now that IMP supports the use of mapped files to provide shared memory, some of this problem has been alleviated, but IMP still needs a number of semaphores and message queues. It is planned to remove these dependencies, at least for Solaris where POSIX IPC mechanisms can be used instead, but for the moment some of what is described here may be needed. This is a job for a system manager.

17.2.1 SunOS

On SunOs 4.1.x machines, you need to modify the file /sys/(arch)/conf/(host)

where arch = result of the command "uname -m", and host is the host specific configuration file. Add or modifiy the lines for the items below. The values shown are those used at AAO; other users may need to modify them.

```
options IPCMESSAGE      # Enable System V IPC message facility
options IPCSEMAPHORE    # Enable System V IPC semaphore facility
options SEMMNI="30"     # No. semaphore ids
options SEMMNS="100"    # No. Semaphores in system
```

The following two lines are only required if shared memory is being used instead of mapped files, and so should no longer be needed for recent versions of IMP.

```
options IPCSHMEM         # Enable System V IPC shared-memory
facility
options SHMSIZE="3072"   # Max shared memory
```

17.2.2 Solaris

On Solaris 2.x (SunOs 5.x) machines, you need to add the following lines to `/etc/system`.The values shown are those used at AAO; other users may need to modify them.

```
set semsys:seminfo_semmni=30
set semsys:seminfo_semmns=100
```

The following two lines are only required if shared memory is being used instead of mapped files, and so should no longer be needed for recent versions of IMP.

```
set shmsys:shminfo_shmmax=3072000
set shmsys:shminfo_shmseg=18
```

17.3 VMS

Under VMS a number of logical names are used for detailed control of IMP, as follows:

Transmitter buffer size The logical name '`IMP_NET_KBYTES`' can be set to some integer value to specify the number of KBytes of memory to be requested by the Transmitter task when it loads. If large network messages are to be sent this will need to be set, since the default value is only 1024 (that is, 1024 KBytes are allocated). This is really only necessary in the absence of the long-promised mechanism where memory is allocated dynamically for new connections.

Location of network startup files The logical name '`IMP_STARTUP`' is used to control where the network tasks look for any startup files. This is described in detail in the section on 'Network startup files'.

Location of network tasks The logical name '`IMP_DIR`' is used to specify the directory where the IMP Master task looks for the network tasks ('Transmitter' and 'Receiver'). If this is not defined, or if the executables for these tasks are not present in the specified directory, the Master task looks in the directory from which it itself was run and then at the current default directory.

Event flag numbers Note: this is pretty obscure stuff, and it's hard to imagine a system where it's an issue. However, it needs to be documented just in case. Generally, an IMP user can ignore this completely. There are two places in the IMP system where it has to make use of an event flag with a specified number. Normally, under VMS, when you need an event flag you get one using the library routine '`Lib$Get_Ef()`', which doles one out from a cache of unused event flags. However the event flag associated with the mailbox used for the X-compatible notification mechanism has to be in the range 1-31 for compatability with the VMS implementation of the X event loop, and '`Lib$Get_Ef()`' allocates event flags from the range 32- 63. The event flag used by the Transmitter task to control its sockets has to be in the same range as the event flag used for the X-compatible notification mechanism. This means that the event flag numbers used in these two cases have to be hard-coded into the IMP system, and there is a remote possibility that there is a conflict with some complex task that uses IMP and also uses specific event flag numbers. Normally, IMP uses event flag 21 for the notification event flag and 22 for the sockets used by Transmitter. If these numbers cause a problem, they can be overridden by setting the logical names '`IMP_NOTIFY_EF`' and '`IMP_SOCKET_EF`' to a string that can be decoded to give the event flag numbers to use instead.

17.4 VxWorks

VxWorks supports environment variables. IMP makes use of the following:

Transmitter buffer size The environment variable '`IMP_NET_KBYTES`' can be set to some integer value to specify the number of KBytes of memory to be requested by the Transmitter task when it loads. If large network messages are to be sent this will need to be set, since the default value is only 1024 (that is, 1024 KBytes are allocated). This is really only necessary in the absence of the long-promised mechanism where memory is allocated dynamically for new connections.

Location of network startup files The environment variable '`IMP_STARTUP`' is used to control where the network tasks look for any startup files. This is described in detail in the section on 'Network startup files'.

18 Current state of play

At the time of writing (6th November 1998), IMP is at version 1.4.2 and the system described in this document has been implemented for VAX/VMS, for UNIX (six slightly different versions exist, for ULTRIX, SunOS, Solaris, Linux, HP-UX and OSF/1), for VxWorks, and for Windows 95/98/NT. The ULTRIX port is no longer kept up to date, as there seems to be no demand for it. However, it could easily be resurrected. The Windows version is still slightly experimental and has not been used much yet in anger.

Almost all the code is system-independent, with two modules called `impz` and `impz_load` containing those routines that have to be implemented differently for the different systems. There are different versions of these for the various systems supported by IMP, so there are for example, files called `impz_vms.c`, `impz_load_vms.c`, `impz_unix.c`, `impz_load_unix.c` etc. The design allows for more than one mechanism to be used to pass messages locally between tasks, the idea being to use whatever proves the most efficient on each system, but at present all the implementations pass messages between tasks on the same system using ring buffers in shared memory (which do not need the additional overhead of semaphores to control access to them) and using a system-dependent notification mechanism (for VMS, for example, this is a 'hibernate-wake' mechanism). Different mechanisms are used when the process in question has to coexist with other systems, such as X-windows (on VMS a mailbox is used, which is significantly slower than the hibernate/wake mechanism).

To see how the system seems to be meeting the criteria listed at the start of this document:

Speed The speed seems reasonably good. The notification mechanisms used are reasonably efficient and the housekeeping overheads involved in actually manipulating the messages are kept as small as possible. However, there is always scope for improvement. Some figures are given below.

Multiple machines IMP has been implemented under: VMS (VAX), Ultrix (DECStation), SunOS (SparcStations), Solaris 2 (SparcStations), OSF/1 (DEC Alphas), HP-UX (HP workstations), Linux (both for Intel PCs and for Macintoshes), VxWorks (680x0 systems) and Windows (both 95/98 and NT). A short discussion about porting it to other systems can be found in the next section.

Networking The system has been tried between processes running on the same machine, on machines running on an ethernet LAN, and over a particularly contorted internet link that introduced delays of about 1.5 seconds per message. It has also been tested over PPP connections, which have the reputation of breaking weak networking code. It all seems to work as required.

Message size The size of message that can be sent by IMP is limited only by the amount of shared memory that can be allocated on the machines in question. Tests so far have used messages up to 20 Megabytes. Really large blocks of data can usually be transmitted more conveniently using the bulk data facilities provided by IMP rather than by sending them as ordinary messages, since these don't require suitably sized message buffers to be pre-allocated.

Non-blocked operation IMP routines don't block. Since dedicated ring buffers are used for messages, either a message can be sent immediately or an error status is returned. Messages to other machines are sent immediately to a transmitter task that will send them on — an interrupt routine can send a message to a task on a remote machine knowing that the

`ImpSend()` call will return immediately. With the advent of version 1.0 large network messages no longer block shorter messages sent on different connections.

Waiting without overheads All the notification mechanisms allow a waiting process to remain dormant. Polling is not used.

Reliability Truly crashproof systems are rare, but a lot of work has gone into making IMP robust, particularly in the cases where network connections to remote machines are lost and subsequently reconnected.

Interrupt level working Appears to work satisfactorily, from VMS AST routines, UNIX signal handlers and from VxWorks interrupt handlers.

19 Performance

Some idea of the IMP system performance can be obtained from the following figures. These figures have been collected over the years, and show the change in machine performance in recent times. The test in question is a simple one based on setting up a two-way connection between two tasks and sending messages between them using `ImpSendPtr()` and `ImpReadPtr()`. Since the pointer versions of these routines are used, the time taken — at least when the tasks are on the same machine — is independent of the message size. (Although, obviously, any real application would have to spend time creating the message in the buffer obtained by the `ImpSendPtr()` call and in processing it at the other end, and some applications might not be able to take advantage of the pointer based routines at all.) The times quoted are for a single message (one send, one receive). The test programs used are not the '`atest`' and '`ttest`' pair of example programs shown in this document, but a trimmed-down pair of programs '`test1`' and '`test2`' that just pass minimal length messages without testing the contents.

Note that the tests involved a 'send message, wait for response, send next message' sequence, so each message is read and replied to before the next one is sent, meaning that a context change from one process to another occurs with each message sent. This is not necessarily a realistic test. Also, a significantly greater throughput can be obtained with IMP using the alternative handshaking methods described earlier. This means that what is being tested here is really the time the system in question takes to do a context switch on the basis of whatever notification mechanism — such as a semaphore — is being used, plus the IMP overheads in sending and reading a very short message. (Some of these timings are for older versions of IMP; it's hard to get access to enough unloaded machines to repeat the tests properly, but I don't think the newer versions are any slower.)

Solaris 2.3 (a SPARCStation LX) : 510 microsec

SUN OS (a SPARCstation IPX) : 375 microsec

ULTRIX (a DecStation 3100) : 270 microsec

VxWorks (a 68030 running at 33Mhz) : 252 microsec

Solaris 2.4 (SPARCStation LX) : 201 microsec

VMS (a microVax 4000/300) : 160 microsec

HP-UX (an H-P model 712/80) : 150 microsec

OSF/1 (an Alpha 1000/300X) : 140 microsec

Solaris 2.4 (a SPARCStation 20/HS22) : 91 microsec

MkLinux DR1 (Macintosh 7300/180) : 76 microsec

Linux (Pentium 166Mhz MMX notebook) : 30 microsec

PPC Linux (Macintosh 7300/180) : 22 microsec

Solaris 2.6 (UltraSPARC with two 360MHz processors) : 20 microsec

PPC Linux (Machintosh G3 233Mhz) : 13 microsec.

By comparison, a similar test on VxWorks on the same machine, using the VxWorks msgQSend() and msgQReceive() for messages of varying length, gave the following times: 1 byte messages, 175 microsec; 80 byte messages, 200 microsec; 2048 byte messages, 625 microsec. IMP is clearly slower for short messages, but its greater flexibility may make up for that. I will continue to try to find where the bottlenecks are and see how much better I can do (I believe most of the overheads are in the reading, rather than the sending of messages.) It's interesting to note how much better Solaris 2.4 was compared to Solaris 2.3. The same machine was used for both tests, and shows that an improved operating system can really make a significant difference. It's also interesting to note that the current champion is Linux, running on someone's home computer. Unfortunately, at the time of writing, I've not managed to find a really fast PC running Linux to play 'Mac vs. PC' with. It looks as if a top of the line Mac (currently 333Mhz) and a top of the line PC (currently 450Mhz) should be about equal, running Linux, both around the 10 microsec. level.

20 Porting to other machines

IMP could not have been coded in a totally system-independent way; the things that it does are rather too low-level for that. (It may be claimed that ADA would have allowed it, but we needed it in C.) However, the amount of system-dependent code it contains is relatively small: the current VxWorks version of the two kernel code files (IMPZ_VXWORKS.C and IMPZ_-LOAD_VXWORKS.C) contain 1460 comment lines and only 671 lines of actual code. The UNIX and VMS implementations are larger, particularly the VMS version which is blown out to 1136 lines of real code by the overheads involved in calling VMS system services from C and by a particularly complex process load routine (which uses either LIB$SPAWN or SYS$CREPRC depending on whether or not a CLI is present). In principle, IMP could be ported to any multi-tasking operating system that provides the following:

- A means of sharing memory between processes.
- A mechanism that allows one task to wait for some sort of notification from another. (Semaphores, signals, hibernate/wake systems, FIFOs, anything like this will do, but the faster the better.)
- A means whereby a routine can be caused to execute as a timer interrupt handler after a given time delay. (On UNIX, `alarm()` would do, although the better time resolution provided by `ualarm()` or `setitimer()` is preferable.)
- A means whereby a task can find out if another task (identified in some unique way) is still running. (This is tricky in UNIX, which is traditionally weak in routines that provide enquiry routines about the system or other tasks — both SUNOS and ULTRIX support `getpriority()` and that is used at present.)
- A means whereby critical resources can be momentarily locked by one process. This really means some form of semaphore mechanism - on VMS resource locks are used, but they're just an elaborate form of semaphore.
- A callable socket library. (Actually, only the network tasks use this, and they could in principle use other networking routines, but TCP/IP through sockets is what IMP has been coded for.)
- A means of loading tasks and determining when they exit or crash (the latter is often the hard part — on VMS it uses termination mailboxes, on UNIX it traps 'death of child' signals, while on VxWorks it requires the addition of a task delete 'hook' routine).
- A C compiler. ANSI preferably, but not necessarily. IMP uses things like 'void', but not many other things that are not in the old K & R book. It uses function prototypes, but these are all in conditional code and are not used if the compiler can't handle them.

All of these are provided by the systems IMP has been implemented on so far. VMS as such does not provide a socket library and the Multinet library (which costs money) is being used on the VAXes at AAO. DEC's UCX (which also costs money) is also supported. The SUN OS can provide the system V inter-process communications routines (shared memory, semaphores etc) but has to be explicitly built with this included.

Porting IMP to different versions of UNIX generally presents few difficulties. The various UNIX versions differ slightly. but this is partly because different compilers are used, partly because SUN OS provides useful routines such as ualarm() which ULTRIX does not.

20.1 System-dependent IMP routines

As an indication of the sort of system-dependent code used by IMP, this is a list of the various system-dependent subroutines in the two modules `ImpZ.c` and `ImpZ_Load.c`. There are a few intermediate-level routines such as ImpZOpenUserNbd() which combine semaphore access and shared memory section creation in a relatively intricate way, but most of these are very simple, mapping fairly directly onto a single system call — although the actual call differs from system to system — and in many cases are no more than a few lines of code each. The task loading routine, `ImpZSpawnTask()`, is the most complicated single system-dependent routine. Generally, the VMS code is the most complicated, since VMS system services are both messy to call from C (needing string descriptors, in may cases) and so flexible that they are fiddly to use. The VxWorks code is generally the simplest.

`ImpZCreRootMsgSpace` Create root message space for a task.

`ImpZCreateMsgSem` Create semaphore for a task's root message section.

`ImpZOpenUserNbd` Open (creating if necessary) the main User Noticeboard.

`ImpZSemTake` Waits for a semaphore to become available and then takes it.

`ImpZSemGive` Releases a semaphore taken by ImpZSemTake.

`ImpZCloseUserNbd` Closes down a task's root message space.

`ImpZSemVal` Returns the value of a semaphore.

`ImpZAccessMem` Map another task's root message space.

`ImpZPid` Returns the process identifier for the current task.

`ImpZDeleteTask` Deletes a task given its process identifier.

`ImpZParentId` Returns the process identifier for the current task's parent task.

`ImpZIPAddress` Returns the Internet address for the current processor.

`ImpZNodeName` Returns the internet node name for the current processor.

`ImpZInitNotify` Initialises a receiving task's message notification mechanism.

`ImpZCloseNotify` Closes down a receiving task's message notification mechanism.

`ImpZLinkNotify` Links a sending task into a receiving task's notification mechanism.

`ImpZUnlinkNotify` Closes a link with a receiving task's notification mechanism.

`ImpZCommsMethod` Returns a code for the message passing method to be used.

`ImpZNotifyTask` Notifies a task that it has been sent a message.

`ImpZAwaitNotify` Waits for notification that a message has been sent.

`ImpZClearNotify` Clears any pending notifications.

`ImpZGetXInfo` Returns information needed by X-windows to coexist with IMP.

`ImpZDetachMem` Cancels mapping of a shared memory segment.

`ImpZDetachSem` Cancels atachment to a semaphore.

`ImpZDeltaTime` Turns a timeout into a suitable delta time.

`ImpZDeltaValue` Interprets a delta time as a number of seconds and microseconds.

`ImpZRootMsgAddr` Returns the current process' Root Message Space address.

`ImpZHostByName` Returns the IP address of a machine given the machine's IP name.

`ImpZUserPort` Returns the Port number to use for TCP/IP connections.

`ImpZGetAbsTime` Adds a delta time to an absolute time, giving an absolute time.

`ImpZCalcDelta` Given a future absolute time, calculates the delta time from now.

`ImpZTimeSince` Given a past absolute time, calculates the delta time to now.

`ImpZTime1leTime` Indicates if one absolute time is less than or equal to another.

`ImpZTime1gtTime` Indicates if one absolute time is greater than another.

`ImpZSetTimer` Sets a subroutine to be called after a specified interval.

`ImpZCancelTimer` Cancels an outstanding timer request queued by ImpZSetTimer

`ImpZTimerInit` Performs any initialisation needed by timer routines.

`ImpZTimerClose` Performs any closedown needed by timer routines.

`ImpZProcExist` Determines if a process with a specified PID exists or not.

`ImpZTranCode` Translates a system-dependent error code into a string.

`ImpZTranExit` Translates a system-dependent process exit code into a string.

`ImpZWaitEvent` Waits for an IMP message or activity on a list of sockets.

`ImpZCloseSocket` Performs any system dependent socket shutdown.

`ImpZSocketErrInit` Attempts to establish a socket error handler.

`ImpZSocketErrClose` Returns to the situiation existing before a call to ImpZSocketErrInit().

`ImpZFileName` Generates a file name from a symbolic name and a file name.

`ImpZGetSymbol` Returns the value of an externally defined symbol.

`ImpZExeDir` Gives the name of the directory from which the current program was run.

`ImpZCloseSpawn` Releases any resources used by ImpZSpawnTask().

`ImpZInitSpawn` Performs any initialisation needed for ImpZSpanTask().

`ImpZSpawnTask` Load a task and set up a routine to be called when it completes.

21 IMP limitations

Few systems are completely bug-free, and IMP is no exception. As it gets used more an more, and in ways different to those originally envisaged, bugs are found and (in most cases) fixed. However, as the system gets used, limitations in the original design emerge and sometimes these are not easy to fix. This short section mentions some of the known limitations of IMP.

21.1 Fixed-size message buffers

Almost every edition of this document (and this is no exception) has mentioned that one of the highest priority modifications to IMP is the introduction of dynamically-allocated message buffers. At present, an IMP task has to specify the size of its message buffers when it calls `ImpRegister()` and if the connections to it use these all up (or if they become fragmented, which is also possible), then no more connections can be made. It means that a task must specify a worst-case estimate for its message buffer size when it registers, and this is not always known.

Actually, although everyone moans when they first realise this, they soon get used to it and just get used to specifying large buffer sizes. I always feel it's a huge limitation in IMP but I don't get many complaints about it. However, it will come. It merely needs the design of a suitable protocol for a task to create new shared memory for the use of additional message buffers and to notify other tasks of its existence. The new routines for handling shared memory sections are in fact a part of this protocol, but the rest still doesn't exist.

In the meantime, huge amounts of data are now best transferred using the new bulk data transfer routines in IMP.

21.2 Multiple threads

IMP cannot be used to communicate between threads of the same process. IMP is used in multi-threaded processes under systems such as Solaris, but with one thread doing all the message reading; ie only one thread can sensibly call `ImpRead()` or `ImpReadPtr()`. This is a limitation of the IMP task model; IMP identifies a task using the doublet of IP address (to identify the machine) and task Pid (to identify the task on that machine). It has no way of identifying a thread within that task. A possible extention to IMP would introduce a 'thread' identifier to the `IMP_TaskID` structure, and this would enable communication at the thread level (although other changes would certainly be needed to make this work properly, the important pre-requisite is to be able to identify a target at the thread level).

21.3 Multiple users

A related question to the multiple threads issue is the question of multiple users. IMP was designed for instrumentation systems, to allow multiple tasks all run by the same user to run as part of a tightly- connected whole. It does not provide a means to write a server task on one machine that multiple clients on a variety of other machines, owned by a variety of other users, can connect to. There was a more detailed analysis of this problem in the 'imptalk' discussion group, but it comes down to the fact that the IMP networking at present does not know how to handle multiple network tasks on a single remote machine (which is what would be required to make this work). There may be ways of implementing this, and it is still being thought about.

22 Installing IMP

Normally, IMP is released as part of the AAO DRAMA section, and the DRAMA documentation should be consulted to find out how to get the whole DRAMA system. (See the AAO web page at www.aao.gov.au). If you want IMP itself, by itself, it can be obtained and installed fairly easily as a standalone system.

At present, the standalone version of IMP can be obtained from the AAO anonymous ftp site, ftp.aao.gov.au, and can be found in

```
pub/local/ks/imp/imp-1.4.2.tar.gz
```

To install IMP, copy this file over (use a 'binary' mode transfer), then

```
gunzip imp-1.4.2.tar.gz
tar -xvf imp-1.4.2.tar
```

This will produce a directory called imp-1.4.2 which will contain all the IMP source files, including all those needed to build the system under VMS and VxWorks, which are considered later. For the moment, let's assume you want to build the system under one of the supported versions of UNIX. For each supported version of UNIX, there is a makefile with an extension specifying the system. At present, these are:

`makefile.sun` This is for SunOS systems, and has been tested under SunOS 4.1.

`makefile.solaris` This is for Solaris systems. It has been tested under Solaris 2.4 and 2.5. It has only been used on SPARC systems; it may work on X86 implementations, but this hasn't been tested.

`makefile.alpha` This is for DEC Alpha systems running OSF/1.

`makefile.ultrix` This is for DecStations running ULTRIX. Support for this has been allowed to lapse, but it could be resurrected fairly easily if required.

`makefile.linux` This is for Linux systems. This is a relatively untested port.

`makefile.hpux` This is for HP systems running HP-UX and has been tested under HP-UX 10.

Select the one appropriate for your system, let's assume it's one called '`makefile.xxxxx`', and copy it so that it becomes '`makefile`'.

```
cp makefile.xxxxx makefile
```

Now you can make whatever bits of IMP you need. The minimum you need is the library '`libimp.a`', and the two networking tasks '`transmitter`' and '`receiver`'. You will need a version of the Ers routines IMP uses for error reporting. The default UNIX makefiles create a standalone version called '`ers_imp.o`' and use this to build programs such as the network tasks. It should be obvious from the makefile comments how to change them to use another version of Ers, such as the standard DRAMA version. You will probably need the IMP master task, '`master`', and you may want some of the utility programs as well. The most straightforward thing to do is to make all of the programs.

```
make programs
```

This will build the following files, some it which are needed for IMP operation, some of which may be useful additions, particularly if IMP programs need debugging.

`libimp_xxxx.a` Where 'xxxx' is the name of the system, eg sun or solaris. This is the main IMP library, which application programs using IMP will have to be linked against.

`ers_imp.o` This is the standalone version of Ers as supplied with IMP. Application programs using IMP will have to be linked against this or some other Ers library, such as the standard DRAMA version.

`transmitter` The IMP network task that transmits messages to remote systems.

`receiver` The IMP network task that receives messages from remote systems.

`master` The IMP system task that supervises IMP applications and loads the two network tasks. This is needed if the IMP task loading facilities

`netmonitor` A utility that can be used to switch on monitoring of all or a subset of network messages. See the comments at the start of `netmonitor.c` for full details.

`netclose` A utility that closes down the IMP networking tasks.

`impdump` A utility that lists the contents of the various shared memory areas used by the IMP system. See the comments at the start of `impdump.c` and `sysprobe.c` for full details.

`impload` A utility that loads and runs a specified program via the IMP Master task.

`cleanup` A utility needed only under UNIX that clears out any IMP resources, including IMP tasks and inter-process mechanisms such as semaphores that can be left around by IMP tasks that did not close down cleanly.

`showimp` A version of `cleanup` that merely lists the IMP resources on the system and can be run safely without risk of disturbing a running system.

`taskclose` A utility that closes down a registered IMP program.

It also builds the following test programs:

`atest` One of the basic IMP test programs. This is used with '`ttest`' to verify that IMP is operating.

`ttest` The other of the basic IMP test programs. This is used with '`atest`' to verify that IMP is operating.

`test1` A minimal IMP test program. This is used with '`test2`' to check and time the basic sending of IMP messages on the same machine.

`test2` A minimal IMP test program. This is used with '`test1`' to check and time the basic sending of IMP messages on the same machine.

`ashare` Another IMP test program. This is used with '`tshare`' to test the bulk data handling facilities of IMP.

`tshare` Another IMP test program. This is used with '`ashare`' to test the bulk data handling facilities of IMP.

`apeek` Another IMP test program. This is a version of `atest` that uses message 'peeking'. It is run in conjunction with `tpeek`.

`tpeek` Another IMP test program. This is a version of `ttest` that uses message 'peeking'. It is run in conjunction with `apeek`.

22.1 VMS

Since the IMP code is available under anonymous ftp from AAO only as a compressed `tar` file, you need a UNIX machine to extract the files even if you only want to run it on VMS. So you get the files as described under UNIX, and you will find that they include a file called '`descrip.mms`' which is an MMS description file set up to build IMP under VMS. If you give the command:

```
make programs
```

you will find that you have built the same programs as described in the UNIX section.

22.2 VxWorks

The IMP release is set up with a makefile that will build the VxWorks version of IMP on a UNIX system that has the VxWorks UNIX development environment installed. The makefile in question is '`makefile.vxworks`' and to build under VxWorks it should be enough to do the following:

```
cp makefile.vxworks makefile
make
```

This builds two files. `imp.o` contains the whole of the IMP system, including all the programs listed in the UNIX section. `imp_ers.o` is the standalone version of the Ers routines supplied with IMP. You may prefer to use the DRAMA version if you have that available. You can load IMP and start the IMP Master task (which in turn starts the networking tasks) with the commands

```
iam ‘‘username’’
ld < ers_imp.o
ld < imp.o
sp ImpMaster
```

23 IMP release history

Versions 0.0 to 0.3 of this document described the IMP system as it was being developed and before it was actually made any use of. Version 0.3 was used for the development and testing of the DITS (Distributed Instrumentation Tasking System) being developed at AAO for the 2dF (2 degree Field) project. The specification was also circulated outside the AAO and some particularly useful comments were received from William Lupton at Keck.

Version 0.4 reflected some of the changes that were suggested as a result of all this. The calling sequences for all the routines were significantly revised, now making much more use of structures as arguments. This resulted in a significant increase in the speed of the system (on most machines – - the improvement under VxWorks was less significant, which probably says something useful about how to use the GNU C compiler for the 680x0 machines, if I can only work out what). The routines `ImpQueueReminder()` and `ImpMessageCount` were implemented.

Version 0.5 follows the introduction of message 'peeking' and the task loading facility. To support task loading, the IMP 'Master' task has now been written.

Version 0.6 is an intermediate version following a number of assorted changes. ERS error reporting has been added to a number of routine, but by no means all yet. This may result in additional error messages being generated when a program using IMP is run. These can be controlled by calls to ERS routines, but it does introduce a functional difference to the way IMP routines work — until now, they have always been quiet and only returned error indications through the inherited status variables. The error codes used now conform to the DRAMA conventions, and there is a file `imp_err.msg` that defines them. Users should note that the calling sequences for `ImpAccept()` and `ImpConnection` have changed to allow additional information to be passed between the routine and the caller. The enquiry routines `ImpMessages()` and `ImpConnectionInfo()` are new.

Version 0.7 is another intermediate version following a number of internal changes aimed at tidying up the ring buffer code, which had been getting more complex as things like 'peek' reads were added, resulting in at least one potential race condition, now eliminated. The testing for new messages was also recoded and is believed to be a little faster now. ERS error reporting is now much more comprehensive, now covering most parts of the system except the VxWorks system-dependent code. The routine `ImpCloseConnect()` is new, and the IMP_MULTI_READ flag was introduced for multiple concurrent calls to `ImpRead()` and `ImpReadPtr()`.

Version 0.8, although another intermediate version, introduces a number of new features. Some are minor, some are fairly significant. The full list of functional changes is:

Translator tasks These are now supported.

Timer facilities `ImpTimeNow()` and `ImpTimeSince()` are now available.

Task descriptions `ImpSetDetails()` and `ImpGetDetails()` are now available.

Loading without Master Tasks may now set the IMP_MAY_LOAD flags and load sub-tasks without needing the IMP Master task to be present.

Version checking Code has been added to return an error code if an IMP task finds that another IMP task on the system has been linked with an old, incompatible, version of IMP. (This used to lead to a number of strange errors; now at least it gets flagged properly.)

VxWorks task names The 'location' part of a task specification such as 'location:name' is now ignored under VxWorks — this was documented earlier but now actually happens.

Process names In `ImpRunTask()` the IMP_PROG_NAME flag must now be set for the process name in the TaskParams structure to be used. If it is not set, the IMP system will invent a process name for systems where one is required.

Peeking while reading The system now allows a program to 'peek' at messages while it still has a read by pointer outstanding. So you can call `ImpReadPtr()` without the 'peek' flag set, and then call either it or `ImpRead()` with the 'peek' flag specified, as many times as required, and you may call `ImpSetAsRead()` as well, all within the brackets of the original non-peeking call to `ImpReadPtr()` and the corresponding call to `ImpReadEnd()`. The previous documentation implied that this could be done, but it didn't work properly. Note that a program that does this sort of thing does not *have* to set the `IMP_MULTI_READ` flag when it registers.

Version 0.9 introduced support for Solaris 2 and for OSF/1. The OSF/1 support, running on an Alpha, was significant because its introduction required a careful review of all the IMP code to see if there were any problems running on a machine where an address was 64 bits and could not be held in an 'int'. There were almost none of these (although there was one subtle problem connected with using an 'int' to hold the difference between two addresses), but there were a number of places where 'int' and 'long' had been used somewhat interchangably and these needed to be corrected. As a result of this, IMP has gone through a very detailed check using lint (which missed the address difference problem, but found most of the the others), which should be good for its reliability. The calling sequences have been revised slightly. The main change is that the status argument used by most routines is now of type '`IMP_Status`' instead of 'long'. Since these types are the same on all the systems used to date (but are different on an Alpha, where '`IMP_Status`' is an 'int') most existing code should still work but may give problems if ported uncorrected to an Alpha. It would be best if all code using IMP were changed to use the correct status type. The other change is the addition of a new parameter to `ImpNetLocate()` so that error messages generated as a result of such a call can be tied in to the call in question. The prototypes in imp.h can probably be used to locate most problems with calling sequences.

Version 1.0 introduced a number of changes to the system. As usual, some are minor, some are fairly significant. The full list of functional changes is:

Flow-control 'Flow-controlled' connections are now supported. The routines `ImpRequest-Notify()` and `ImpCheck Send()` have been introduced to support this facility. The new routine `ImpAcceptConnect()` has been introduced to allow a flow- controlled connection to be accepted properly. This routine is a replacement for `ImpAccept()` which allows a set of flags to be specified. `ImpAccept()` is still provided and will not be removed, but it is no longer needed, since `ImpAcceptConnect()` provides a superset of its capabilities.

Startup files Startup files for the network tasks are now supported.

Fully asynchronous networking The network tasks have been completely rewritten to operate completely asynchronously, meaning that long network messages are no longer allowed to block short ones sent on different connections. The code for the network tasks has been properly commented, at last (all the more necessary, considering how much more complicated asynchronous code can be).

Network buffers By default, the Transmitter network task registers with a default of 1 MByte of allocated memory. This can now be overriden by setting the IMP_NET_KBYTES symbol (logical name or environment variable) to the number of KBytes needed. If large messages are to be sent from machine to machine this will need to be defined.

Network monitoring There is now rather more control provided over network message monitoring (as invoked by the NetMonitor utility). This feature — and the NetMonitor utility itself — is used mainly during program development, and may not be of much interest to general IMP users.

Use of `mmap()` On UNIX systems that support the `mmap()` file mapping call, this is used instead of the System V shared memory mechanism (`shmget()` and related routines) to provide shared memory. This means that SunOS and Solaris systems no longer need to be configured with large amounts of System V shared memory in order to run IMP. The IMP_SCRATCH environment variable is used to control where shared files are located. (Things work much better if these files are on a locally mounted disk!)

Bug fixes A number of bugs have been fixed; in particular, there should now be fewer cases where tasks on VxWorks systems remain registered on UNIX systems even when the VxWorks system has crashed. The system can now spot an out of date user noticeboard and can clear it. Some cases where a ring buffer was being used even though it was no longer current have been trapped.

The following note was added to the version of this document that appeared with version 1.0 of IMP:

"There is nothing particularly significant about the use of '1.0' as a version number. It doesn't mark any particular milestone, or the final appearance of a version regarded as fit for general consumption instead of purely internal use. It just happens to be the next after 0.9 (aproaching 1.0 asymptotically via 0.91, 0.92 etc seems to indicate a lack of confidence in the system!). However, by now IMP is a reasonably mature system and I am quite happy to see a 1.0 version go out. There have been a number of changes to the code, particularly in the network tasks, and these may well introduce bugs that only show up when users outside AAO make use of the system. If that happens a 1.1 bug-fix version will be released quickly to fix these."

Version 1.1 was, as it turned out, something quite different. As the result of a short visit to ESO, a version called 1.1 was released there. This included support for HP-UX, and the first release of the bulk data routines (`Imp SendBulk()`, `ImpReadBulk()`, etc.). This version was put together in rather a hurry, and was lacking a number of necessary features, such as VxWorks support for the bulk data routines, so it was not generally released.

Version 1.2 was intended to be the first generally available release of IMP with support for the bulk data routines. As it happended, it wasn't properly released; it escaped, in a number of beta versions, but the one thing missing from most of these was any up to date documentation. Enough versions with a 1.2 version number have appeared in this process that it now seems worth giving a new number to the release put out at the beginning of June 1997 and calling it 1.3.

Version 1.3, therefore, is a consolidation of the various changes made to IMP since 1.0, and is the first properly documented version to go out since 1.0. The following list summarises the various changes made to IMP between the releases of 1.0 and 1.3. Briefly, there are a few new features, mainly the bulk data facility, the enhanced startup files, and support for HP-UX and for Linux. There are also a number of bug fixes, mainly connected with making sure that tasknames are correct in connection messages and in error messages, and a significant amount of work to remove memory leaks from the VxWorks implementation.

HP-UX IMP now supports the HP-UX version of UNIX. This was introduced with version 1.1 and improved in the later versions.

Linux IMP now has support for the Linux operating system; this appears to work both on the 'conventional' PC implementations and on the experimental MkLinux version, but is relatively untried.

Bulk data routines The routines `ImpSendBulk()`, `ImpReadBulk()`, `ImpHandleBulk()`, `Imp-BulkReport()`, `Imp DefineShared()` and `ImpReleaseShared()`, which between them comprise the bulk data handling system, all were introduced with version 1.1. These routines are now supported on all IMP platforms.

Bad local status in `ImpReadBulk()` Until version 1.3, `ImpReadBulk()` was ignoring the value of the `LocalStatus` argument, meaning that if a target task failed to map memory to receive the bulk data there was no way it could make the sender aware of this. This is fixed in 1.3.

'`showimp`' and '`cleanup`' The '`showimp`' command has been added, as a harmless way of invoking '`cleanup`' in its report-only mode. A minor reformatting of the output from '`cleanup`' should make it easier to see what has been done. The '`cleanup`' command now checks for (and can delete) the Fifo files generated in `/tmp/ff.*` by the Fifo notification mechanism.

`ImpSysProbe()` This routine replaces `ImpProbe()`. Although the latter is still included, it it now implemented through the more general `ImpSysProbe()`. This now accepts UNIX style options controlling the amount of information output. Some additional information, particularly a list of processes that have mapped the message space of the listed processes, has been added.

'`impdump`' The '`impdump`' utility now uses `ImpSysProbe()` and so accepts the same set of options for controlling the information output. The most common invocation is probably '`impdump -t taskname`' which restricts the information output to that relevant to the named task.

File mapping checks On systems that support the '`statvfs()`' system call, IMP now checks that sufficient disk space is available before it creates a file for shared memory access through file mapping. Some systems (HP-UX) allow files to be created apparently successfully even though the disk is full; this disk space check side-steps this problem. On systems that support the `mincore()'` system call, the memory address space allocated to a mapped file is now tested for validity after the mapping; this provides yet another safeguard.

Memory leaks The VxWorks version of IMP has been fairly thoroughly investigated for memory leaks. There were some huge ones — the message buffers for processes were not being deleted on exit, for example — which meant that systems that had a number of small processes constantly being created and then deleted would leak memory very badly. It is now believed that all memory leaks in VxWorks (and this should mean other systems too) have been plugged.

Port numbers The port number used by the IMP networking tasks can now be specified in the startup files.

Testing connections The startup files now allow you to specify that connections to specific machines should be tested at regular intervals. This should allow more reliable detection of the case when a VxWorks system crashes and restarts, but note that it has to be enabled explicitly — the default is not to test connections in this manner. This is a much better way of testing for a crashed VxWorks system than setting the VxWorks system to connect on startup to a UNIX machine — see the section on startup files for more details.

`ImpTaskLocal()` This is a new routine that allows a program to determine whether or not a specified task ID refers to a task on a local machine. This is more elegant than explicitly testing the IP address held in the task ID structure, and much less likely to break in later releases of IMP.

OpenVms A number of changes to the VMS-specific code have meant that it now works under OpenVms, and with the Dec UCX library as well as with the Multinet library used at AAO.

`ImpErrorText()` This is a new routine that returns a pointer to the error text associated with an IMP error code. Use of this makes for much more readable error messages.

'Standalone' **`Ers`** IMP 1.3 contains a standalone version of the subset of Ers routines used by IMP, based on code originally provided by Allan Brighton at ESO. This makes it easier to build IMP on systems that do not have the usual DRAMA- supplied version of Ers.

Makefiles The makefiles supplied with IMP have been modified to link against the standalone version of Ers, and no longer need the DRAMA startup to have been run. There is now a makefile equivalent ('`descrip.mms`') provided for VMS.

EAGAIN errors The UNIX code now tests in places for system routines returning the EAGAIN error code to indicate a transient error, and attempt to retry the system call in question. This is needed because SVR4 systems in particular are prone to return this error when heavily loaded.

Tasknames and errors A number of problems, particularly with DRAMA's use of IMP, were investigated and traced to problems with IMP error messages not properly identifying the tasks involved. In particular, 'connection closed' messages were being generated with very misleading task names and identifiers, although these were not the only cases where tasks were misidentified in IMP system messages. The case where a two-way connection failed because the originating task ran out of message buffer space, meaning that the final connection back to it failed, was investigated particularly thoroughly, and revealed a number of errors in the IMP error handling. The case where the IMP Master task was shutdown, leading to a breakdown in network communications, was also a rich source of bug fixes. All known cases of such errors have now been fixed in 1.3, although there is no guarantee that others are not still lurking in the code.

All machines lost A special case of the system message classified by `ImpSystemMessage()` as IMP_SYS_MACHINE_LOST has been introduced. This specifies the IP address of the missing machine as -1, and indicates that contact with *all* machines on the network has been lost. The routine `ImpLostMachine()` has been modified to allow for this case, and since most code that uses IMP will use this rather than test for explicit IP address, this should be a fairly transparent change. However, it just may introduce problems for some rather esoteric code.

Outdated user noticeboard The message about an 'outdated user noticeboard', generated when IMP determines that although a user noticeboard exists all the information it contains refers to tasks no longer in the system, is no longer generated by IMP 1.3. It was found to be unnecessarily confusing. The test and the clearing of the noticeboard still occur; only the warning message has been suppressed.

'`impload`' The '`impload`' utility now concatenates its arguments properly, with spaces between them, and tests for internal buffer overflow if the argument string generated is too long.

Revision number Changes to the network messages and to the various structures used internally by IMP have meant that code using IMP should be re-linked; code linked with 1.3 may not run properly with code linked against earlier versions. The revision number used to catch such mis-matches has been incremented accordingly.

Documentation This document itself has been substantially revised for version 1.3. It had become extremely bulky, and as a result the source of the various example programs has been removed, and the appendix with the individual routine comments has been separated off. This is all now available on the Web at www.aao.gov.au.

Version 1.3.2 was an intermediate version, introduced relatively soon after 1.3, that fixed a number of bugs in 1.3, mainly to do with handling broken connections (caused when a task or a remote machine went down) cleanly. One thing introduce by this version was the ability to log messages and network events, which proved a very useful diagnostic tool. The ability to test network connections using the 'Pulse connections..' startup option was added. A separate document is available describing in detail the changes made between IMP 1.3 and IMP 1.3.2

There were a couple of interim versions of IMP used internally at AAO. These acquired the version numbers 1.4 and 1.4.1, but were never made generally available. IMP version 1.4.2 is the version released for general use in November 1998.

Version 1.4.2 contains a large number of changes from 1.3.2, although these have mainly been aimed at improving the reliability of the system. The network tasks now have a much tighter handshake after each message. This eliminates a number of problems that were connected with the amount of message buffering in the socket layer, making it hard for IMP to know which messages had actually been received by a recently crashed remote machine before it went down. A number of problems connected with identifying which tasks used which connections, which could complicate recovery when a machine or a task went down, were all fixed. Message delivery in general is much more tightly controlled — for example, when a task crashes, the system can now spot messages in its buffers that had been delivered but not yet read, and the sending tasks are now notified of these unprocessed messages. Very few new features were added. The full list of new features is as follows:

Multiple IP addresses Code to handle machines with more than one network address has been added. The 'Note alias...' startup option is now provided to support such machines.

Data associated with bulk data Bulk data transfers now allow associated data to be specified and transmitted at the same time as the bulk data transfer is initiated.

Simplified bulk data protocol Tasks can now register with the IMP_FORCE_READY flag specified, and they will only be notified of the arrival of bulk data when its transfer is complete and the data is ready to use, already mapped on the local machine.

Windows support IMP now supports the Win32 API, and IMP programs will now run under Windows 95/98 and Windows NT. This implementation is still relatively untried.

Version 1.4.3 had a number of minor changes, mainly bug fixes, particularly in the area of VMS networking.

Version 1.4.4, released at AAO in March 2000, has a few more bug fixes, and one new feature, namely support for the 'use address' network startup option.

24 Example programs

At this point, earlier versions of this document included the source code for some relatively simple example programs using IMP. These were the programs '`atest`', '`ttest`', and '`ttran`'. However, there seems little point in including these now, as the code is included in the IMP release, and with time this document has grown to the point that the last thing it needs is additional padding. So the code has been removed. However, any interested readers can always print off the files '`atest.c`', '`ttest.c`', and '`ttran.c`' and take a look at IMP in action.

'`atest`' and '`ttest`' together form a test of the IMP system used to measure how fast it can send messages between tasks using the simplest – but not the fastest – form of handshaking, namely a protocol where a task sends a message to another and waits for the other task to respond to it before sending the next message. Note that other, faster, mechanisms are available in IMP. The '`atest`' program locates a second task, establishes a two-way connection to it, and starts to send messages to it, waiting after each message for a response from the target task. The '`ttest`' program is a suitable target program, simply responding to messages sent to it. A single copy of '`ttest`' can handle a number of simultaneous copies of '`atest`', since all it does is read a message and send it back to its sender, then read another message. '`ttest`' is by far the easiest of the two to understand, it you will probably find it best to look at it first and then at the code for '`atest`'.

The structure of '`atest`' may be interesting, since it has been written as a program that is 'event-driven', responding to messages by calling action routines appropriate to that message type, and maintaining an internal state structure that enables it to control the way the action routines respond to the messages that have caused it to be called. Even in a program of this size, this was found to provide a more flexible structure for dealing with the messages that a program using IMP can find itself receiving.

These two programs comprise the initial test suite always run after changes have been made to IMP to verify that the basic system is still running. I always build these two, then run them, first on the same machine. Typical output from the test is shown below. 'aaossi' is the name of the system being used. '`ttest`' is started first. It prints out its message buffer size, then registers under the name specified in its command line ('fred' in this case) and waits for connections. Then '`atest`' is run with the command line specifying the local machine 'aaossi', the registered name of the task running '`atest`', ie 'fred', and the number of messages to be sent.

```
aaossi > ./ttest fred &
[1] 3355
aaossi > Message size = 3072
ttest registered OK
aaossi > ./atest aaossi fred 100 &
[2] 3356
aaossi > Registered OK
Located task 'fred' OK
Connect request from task 'Proc 3356', pid 3356, machine c0e7a70d
Connected OK to task 'fred'
Timer starting
100 messages sent in 0.051488 secs
ie 514.880005 usec per acknowledged message
Maximum time for any acknowledged message = 157.000000 usec.
(remember that the system clock may have quite coarse resolution)
[2]    Done                   ./atest aaossi fred 100
```

```
aaossi >
```

The machine in question here is a quite heavily loaded Sparc 10 running Solaris 2.4.

The other example program is the simple ‘translator’ program ‘`ttran`’ mentioned earlier in the manual. This is a version of the example program ‘`ttest`’ introduced in the previous section, but written as a translator program. It is intended to be used with the other example program from mentioned above (‘`atest`’) but in this case when ‘`atest`’ is invoked the target task name may be anything at all — if ‘`ttran`’ is running it will accept messages for that task and reply to them, just as the normal ‘`ttest`’ program would have done if started up and given that target task name.

25 Additional information

IMP documentation is held on the Web at AAO. Go to www.aao.gov.au and use the search facility provided to look for IMP. There is a discussion group conducted through a MajorDomo distribution list for IMP. You can subscribe to this by sending a mail message to "major-domo@aaoepp.aao.gov.au" containing the line

```
subscribe imptalk
```