ICIMACS Messaging Protocol

Version 2.5

(IMPv2.5)

Version:	1.2.5
Date:	2008 November 26
Prepared by:	R.W. Pogge & J.A. Mason, OSU

Distribution List

Recipient	Institution/Company	Number of Copies
Richard Pogge	OSU	1 (file)
Jerry Mason	OSU	1
Ray Gonzalez	OSU	1

Document Change Record

Version	Date	Changes	Remarks
0.1	2008-11-20	Draft Spec	
0.2	2008-11-26	Mason & Gonzalez Comments	

Contents

1 Introduction

1.1 Scope

This document describes IMPv2.5, version 2.5 of the ICIMACS Messaging Protocol (IMP) that is used by OSU instruments for interprocess communications. It supersedes the original ICIMACS Messaging Protocol (IMPv1) and introduces a major extension to IMPv2. A protocol is a formal set of rules describing how to transmit data, especially across a network. Low-level protocols define the electrical and physical standards to be observed, bit- and byte-ordering and the transmission and error detection and correction of the bit stream. High-level protocols deal with the data formatting, including the syntax of messages, the terminal to computer dialogue, character sets, sequencing of messages etc.

Except for the adoption of the standard ASCII character set, IMP makes no specification of the programming language or operating system to be used to implement it, nor does it specify the communications medium to be used to send and receive messages between processes. This makes it usable by and between any systems that use ASCII characters to represent text, and extremely flexible with regards to choice of the interprocess communications medium (so far we have used it with RS232 serial ports, TCP/IP sockets, UDP/IP sockets, and Unix FIFO pipes in deployed systems, and experimentally with Unix Domain Sockets, Remote Procedure Calls, and IR-based serial communications). Currently only UDP and RS232 are actively employed in current-generation instruments.

1.2 Reference Documents

1. *Definition of the Flexible Image Transport System (FITS)*, NOST 100-2.0, 1999 March 29 [NASA/Science Office of Standards and Technology, Goddard Spaceflight Center]

1.3 List of Abbreviations and Acronyms

ICIMACS	Instrument Control & IMage ACquisition System (OSU data-taking system)
IMP	ICIMACS Messaging Protocol
IP	Internet Protocol (network layer of TCP/IP)
IPv4	Internet Protocol version 4 (current version of IP)
IPv6	Internet Protocol version 6 (candidate to replace IPv4)
ISIS	Integrated Science Instrument Server
LBT	Large Binocular Telescope
LBTO	Large Binocular Telescope Observatory (operational arm)
LBTPO	Large Binocular Telescope Project Office (administrative arm)
MODS	Multi-Object Double Spectrograph
OSU	The Ohio State University
TCP	Transmission Control Protocol (connection-oriented protocol layered on IP)
UDP	User Datagram Protocol (connectionless protocol layered on IP)

2 Overview of the ICIMACS Messaging Protocol

ICIMACS is an acronym for **I**nstrument **C**ontrol and **IM**age **AC**quisition **S**ystem, the generic name for the suite of systems and programs used to operate OSU's astronomical instruments. The ICIMACS Messaging Protocol is a simple, lightweight, text-based, messaging protocol describing the syntax by which command and status messages are passed between processes running in a "network" of ICIMACS hosts. Messages are formatted to be both human- and machine-readable.

The first version of the ICIMACS Messaging Protocol (henceforth IMPv1) was implemented in 1995 on DOS-based computers with the first post-FORTH generation of OSU data-taking systems. The protocol began to evolve, taking on new functions, and accreting or ejecting old features, until it stabilized into the messaging protocol used in the data-taking systems of all OSU instruments deployed between 1995 and 2002.

The purpose of IMPv2 was to extend the original protocol to handle the more sophisticated next-generation instruments being designed and built by OSU, and to formally define some of the best-practices that emerged as the original protocol developed. The need for a formal revision of IMPv1 emerged in 2003 with the development of a prototype message passing server for use with the MODS spectrographs being built for the Large Binocular Telescope.

The principal enhancements of IMPv2 are as follows:

1. Expanded the node address name sizes from 2 to 8 characters to permit greater flexibility in naming system nodes, and greater legibility of message address headers in large multi-node systems with complex instruments (like MODS).
2. A new message type, EXEC, to provide data-taking system applications a means to lock out sensitive functions from remote execution unless specifically qualified as an "executive override" command request.
3. Formal definition of the implicit "request" (REQ) message type.
4. Formal definition of handshaking as an "out-of-band" message.
5. Introduction of a heartbeat mechanism for monitoring the health of client nodes in complex systems.
6. Formal definition of rules for handling out-of-protocol messages.

IMPv2 is the messaging syntax used by the new OSU data-taking system. The first implementation of IMPv2 with ISIS was the data-taking system deployed with the ANDICAM instrument on the CTIO 1.3m in January 2003. This was really an IMPv1.5 "mixed" implementation, as key nodes in the system (the IR and CCD array control computers and the coordinating "workstation" computer) were still IMPv1 systems running under DOS. A second IMPv1.5 system was deployed with OSIRIS at the SOAR 4.2 meter telescope in early 2004. The first full IMPv2+ISIS system was created for the Y4KCAM deployed at the CTIO Yale 1-meter telescope in March 2004.

With the MODS instruments for the LBT we have the first system that allows concurrent operation of instrument mechanisms and have a much larger number of subsystems than previously (nearly an order of magnitude greater system complexity). IMPv2.5 addresses incremental changes to IMPv2 that address these challenges.

3 ICIMACS Messaging Protocol version 2.5

3.1 Message Format

3.1.1 Basic Syntax

IMPv2.5 message strings take the following form:

```
srcID>destID msgType cmdWord msgBody\r
```

where:

`srcID`	name of the sending ("source") node (8 chars max)
`destID`	name of the destination node (8 chars max)
`msgType`	type of message being sent to `destID` (§3.3)
`cmdWord`	command word (character string w/o spaces)
`msgBody`	command arguments or message text passed to `destID` (§3.4).
`\r`	termination character ("carriage return" = ASCII 13)

Each of the message string tokens are separated using the ASCII space character (ASCII 32). Multiple spaces are ignored. Within `msgBody` spaces are used as token separators for arguments (either command arguments or parameter keyword=value pairs reporting return status), or as part of the text messages. This syntax is generally the same as used by IMPv1, but occasionally differs in essential details as will be described in the following sections.

The most significant change between IMPv1 and IMPv2 is the requirement that all message strings include the command word that the message is being sent in response to.

3.1.2 Character Set

IMPv2 message strings must only consist of printable ASCII characters, of both cases, and contain no non-printing characters except the terminator. Null characters (ASCII 0) must not appear anywhere inside a message string.

3.1.3 Message Length

The maximum message size, including the terminator character, is 2048 characters.

3.2 Address Header

The address header of an IMPv2 message consists of the sending and receiving node names separated by a > character (ASCII 62). **An address header must be the first component of a valid IMPv2 message.** Extraneous leading spaces may be ignored, but no other characters may appear before the address header.

3.2.1 Node Names

Nodes names are "logical addresses" that are mapped onto physical addresses (e.g., serial port devices, IP addresses, etc.) by the individual applications. Node names must be composed of ASCII characters, 8 characters maximum, and 2 characters minimum. Node names are **case insensitive** and composed only of letters [`A-Z`], numbers [`0-9`], and the special characters "**.**" and "_". No other characters may appear as part of a node name.

In IMPv1, node names were 2 characters long. IMPv2 allows up to 8 characters maximum, 2 characters minimum. Single-character node names are formally forbidden by the protocol.

3.2.2 Node-name separator character

The > character (ASCII 62) is used as the node-name separator and reserved. No spaces may appear on either side of the separator.

3.2.3 Reserved Broadcast Node Name "AL"

The node name "AL" is reserved as the broadcast address. An IMP-compliant routing message handler receiving a message addressed to node "AL" must pass the entire message to all known non-broadcast nodes. AL is short for ALL, and IMPv2.5 retains the 2-letter syntax for backwards compatibility with IMPv1 systems. "ALL" will be recognized by IMPv2.5 systems as an alias for AL.

3.3 Message Types

Message types tell the receiving application how to process the message. There are 7 message types used with IMPv2:

Type Code	Means that the message text following is a(n) …
REQ:	command request [implicit if omitted]
EXEC:	command request with executive override priority
DONE:	command completion acknowledgment
STATUS:	informational status report (e.g., command progress)
ERROR:	error message: the requested action terminated with errors
WARNING:	warning message: an unexpected result has occurred
FATAL:	severe error condition that precludes acquiring data

3.3.1 Command Request (REQ:)

REQ: is as an explicit message type starting with IMPv2. It is the default "implicit" message type to be assumed by the receiving node if none of the other Message Type codes appear after the address header (§3.2), the default behavior in IMPv1. This allows IMPv2.5 to retain the convenience of being able to omit REQ: for commands sent by hand from an interactive command-line interface.

All REQ: type command requests are two-way in the sense that an appropriate reply must be sent back to the original sending node upon completion of the request (or upon failure to execute or complete same because of a fault condition). Replies must be of the DONE:, STATUS:, or error message types as appropriate.

3.3.2 Executive Command Request (EXEC:)

The EXEC: was introduced with IMPv2 and is designed to allow applications to define a subset of "expert" commands that are not to be executed by a remote host unless sent as an explicit EXEC: command requests. For example, a data-taking system application might be written to only accept a "QUIT" command (to terminate application execution) from that application's interactive command-line interface but otherwise ignore a QUIT sent from by remote node unless it is sent as EXEC: command request. Thus EXEC: class messages are a subset of REQ: class messages, and can be interpreted as "executive override", meaning it allows the remote application to override the normal execution restriction.

All EXEC: type command requests are two-way in the sense that an appropriate reply must be sent back to the original sending node upon completion of the request (or upon failure to execute or complete same because of a fault condition). Replies must be of the DONE:, STATUS:, or error message types as appropriate.

3.3.3 Command Request Completion Acknowledgement (DONE:)

A DONE: message signals to the recipient that a command request (implicit REQ: or EXEC:) has been successfully completed with no errors. All DONE: messages are terminal in the sense that they close a command transaction and do not require the recipient to send an acknowledging message in response. Formally, a command request is not complete unless a DONE: or error message is received. See STATUS: (§3.3.4) messages for use when reporting progress of a command still executing.

DONE: messages are accompanied by completion status information, including keyword=value pairs of essential parameters set/changed as part of the command request, and often include human-readable text describing the command-request completion.

If a command request has completed with warnings or aborted prematurely with errors, one of the error message types (§3.3.5) are to be used, depending on the precise fault condition.

3.3.4 Informational Status Messages (STATUS:)

The STATUS: message type is used to transmit status information regarding a command request in progress but not yet completed. STATUS: messages may also be sent out to inform recipient nodes of changes in status of various functions, for example to signal when a node is going offline, or when a node's state has changed. Such messages may be broadcast or single-cast as required. STATUS: messages are one-way in the sense that they do not require the receiving node to send an acknowledging message in response.

Formally, a command request is not complete until a corresponding DONE: is received, so STATUS: messages must not to be used to acknowledge completion of command requests, even though the detailed message body may be nearly identical in contents.

If an error occurs, the error-message types (§3.3.5) are to be used. STATUS: message types must never be used to convey error or warning conditions.

3.3.5 Error Messages (ERROR:, WARNING: and FATAL:)

These message types are used to signal that the command request has terminated with an error, with the severity of the error indicated by the message type chosen. Error messages may also be broadcast to all processes to indicate that some condition requiring attention has occurred. All error messages are one-way in the sense that they do not require the receiving node to send an acknowledging message in response.

The error message types are as follows in order of increasing severity:

WARNING: messages are sent to signal that anomalies have occurred that should be noted, but otherwise the command request is still in progress. Warnings may also be broadcast to inform nodes that an unusual condition, possibly requiring attention, has occurred. A

WARNING: message is thus a special type of STATUS: message. If a warning is generated during the course of command execution that otherwise completes, a subsequent DONE: message is required to ensure that the command request transaction correctly terminates. Similarly, if a warning is followed by the command request aborting with errors, an ERROR: or FATAL: message (as appropriate) is required to terminate the transaction.

ERROR: messages are sent when a requested command aborted with an error that prevented successful completion. Specific instances include

Syntax Errors: the command requested and/or its arguments are unrecognized

Validation Errors: one or more command arguments are invalid (e.g., a requested position is out of the allowed range).

Execution Faults: the command request and its arguments were valid, but a fault occurred that aborted execution before completion (e.g., a mechanism is unavailable, busy, did not respond within some reasonable timeout interval, or failed to perform the requested action).

FATAL: messages are sent when a requested command results in an error of sufficient severity that physical intervention is required to clear the fault. Nodes receiving a FATAL: error message, if so configured, should abort whatever procedures they are executing and enter a "safe mode" to await user intervention. This error class is specifically reserved for "really bad things", like failure of an array controller during an exposure which obviates continuing the observation until there is physical intervention to correct the fault condition.

3.4 Command Word

The `cmdWord` string follows the message type code, providing the context for the message.

This required syntax item is new to IMPv2.5 and was introduced to make command transaction synchronization explicit. This requirement emerged as our systems have begun to migrate from earlier systems in which system nodes were explicitly serialized command/reply systems, to those that permit command concurrency (e.g., an instrument that can move multiple mechanisms at a time).

3.5 Message Body

The `msgBody` follows the `command` string and contains the essential information of the message.

Command requests are usually posed in "command arg1 arg2 …" syntax familiar from most command-line interpreters. For REQ: and EXEC: messages `msgBody` is interpreted as "arg1 arg2 ..." for the command string.

Command-completion (DONE:), informational status (STATUS:), and error messages may consist of unstructured but otherwise human readable text (e.g., `ERROR: filter Requested filter position 42 is out of range: must be 1..12`), and/or a set of formal, easily parsed keyword=value pairs to convey specific information to the receiving process (e.g., `DONE: filter filtpos=12 filtname='SDSS u'`).

Valid key=value pairs take the following forms, depending on the data type. The basic structure resembles that of FITS header keywords, except that no intervening spaces are allowed because spaces are used a token separator between adjacent key=value pairs.

3.5.1 Numerical Values (integers and floats)

Integer and floating-point parameters are specified as numbers without spaces surrounding the = sign. For example:

```
Filter=3
Current=3.30
```

Optional units may be appended using a space separator.

3.5.2 Boolean (logical) Values

Boolean (True/False) values are specified using the T or F characters. For example:

```
ENABLED=T
Open=F
```

As with all keyword=value pairs, the T/F characters should be treated as case-insensitive to aid in parsing, thus (T,t,F,f) should all be valid and interpreted the same.

3.5.3 Character Strings

Single-word character string values are denoted as simple keyword=value pairs with no delimiters. Examples:

```
MODE=TEST
RA=01:14:15.5
HostName=osiris.ctio.noao.edu
```

Note that sexagesimal quantities (e.g., RA, Dec, Time, etc.) are handled as strings.

Multi-word strings (words separated by spaces) must be encapsulated within single quotes (' = ASCII 39) or alternatively in ()'s to delimit the string in keyword=value scope. For example:

```
Object='NGC1068 long-slit R=2000'
Observer=(Pogge, DePoy, and Mason)
```

are valid multi-word string representations. The () delimiters are a legacy from IMPv1 and retained for backwards compatibility. Parsers deployed by compliant applications must be able to handle both instances of multi-word strings.

3.5.4 State Flags

State flags are often designated using a +/– syntax, as follows:

```
+ADDFITS
-VERBOSE
```

Where + is interpreted as "on/enabled" and – is interpreted as "off/disabled".

3.6 Message Termination

All messages must be terminated by a "carriage return" character (CR, Ctrl+M, ASCII 13), or "`\r`" in common parlance (e.g., C, C^{++}, Perl, etc.).

No other commonly used terminator characters (e.g. NUL = ASCII 0 = \0, or NL = ASCII 10 = \n) may appear as part of a valid IMPv2 message string. Messages containing these characters shall be construed to be "malformed", and thus "out-of-protocol" message. See §5 for details on handling out-of-protocol messages.

4 Out-of-Band Messages

A special messaging syntax for "out-of-band" communications is provided with IMPv2, both formalizing a practice instituted in the earliest version of IMPv1 and introducing a new heartbeat mechanism. Out-of-band messages must have a valid address header, but do not use Message Type flags (§3.3).

We are using the term "out-of-band" in its standard computer science sense as data that looks to the application like a separate stream from the main data stream, in our case, meaning outside the standard "command/response" stream. Physically, IMPv2 out-of-band messages use the same physical transport (serial, Ethernet, etc.), and are accorded no higher priority or urgency with regards to utilization of the transport or handling software. They are, however, structured differently enough from the standard command/response syntax that they require separate handling by applications. For reasons of simplicity and transparency, we want to strictly limit out-of-band messages to a very small number of special circumstances.

4.1 Software Handshaking: PING/PONG

The PING and PONG messages are used to perform software handshaking between two nodes. The syntax for PING and PONG are retained as-is from IMPv1.

PING is the "I am here" message used to initiate software handshaking. For example, on startup a client node might send a PING to a server node to introduce itself, or broadcast a PING to all nodes on a message-passing network.

PONG is the acknowledgement of a PING and completes the handshaking transaction. All PONGs are terminal in the sense that they must never generate an acknowledging message in response. In some systems, a PONG message may be accompanied by additional information used to initiate further layers of handshaking. This is optional and receiving node parsers should be able to ignore any extraneous information appearing after the PONG string.

4.2 Process Heartbeat Messages

IMPv2 introduced a simple heartbeat system consisting of a bare address header with no message type or message body. For example:

```
tcs>isis\r
```

is used by the client node named "tcs" to inform a server node "isis" that it is still online. The heartbeat must be properly terminated (§3.6).

Such a bare header would generate a syntax error under IMPv1. All compliant applications must at least handle receipt of bare address headers as non-error conditions, even if they do nothing else with the information.

5 Out-of-Protocol Syntax (OoPS)

Any message shall be considered "out-of-protocol syntax" (OoPS) if it violates any of the syntax requirements described in §0. This section describes how compliant applications should respond to receipt of OoPS messages to prevent problems.

5.1 Malformed Messages

Malformed messages are those that have most of the components of a valid message string but lack a proper address header or a proper terminator character (\r = ASCII 13).

A malformed address header would be one missing the address separator (>), or any characters that are not [A-Z], [0-9], "." and "_", or extraneous spaces *inside* the address header, or lacking one or both node names.

Compliant applications should be written to ignore any message with a malformed address header. For diagnostic purposes, a common practice is to log such messages in a runtime log (if used), but not reply to such messages or echo them to the application console, unless running in a "verbose" or "debug" mode. Under no circumstances should an application broadcast an "ERROR:" message in response to a malformed message.

Compliant applications may elect to treat \n as \r (i.e., implicit translation), much as is done, for example, by the standard Unix /dev/tty interface for serial port communications. While \n is the typical line terminator used in the Unix world, \r is the standard line terminator for ANSI serial communications and so adopted here.

5.2 Extraneous Messages

An extraneous message is any that does not conform in any way to the required syntax. For example, ASCII text received on a serial interface from a non-compliant application (e.g., an application accidentally opens a port it shouldn't have). In these cases, no exceptions can be made. The messages must be treated as "extraneous" and ignored by the IMP-compliant application, except insofar as the messages might be logged or echoed for diagnostic purposes. Under no circumstances should an application attempt to interpret or broadcast an extraneous message.

If communication with devices that do not create compliant messages is required, an appropriate "filter application" should be written that buffers communications with the non-compliant process and translates it into IMP-compliant messaging syntax. Such "agent" applications are commonly deployed with OSU data-taking systems, for example as is done with the PC-TCS interface agent application that buffers and filters telescope pointing telemetry emitted down a serial interface from the PC-TCS system at the CTIO 1.3m and 1-m telescopes.

5.3 Oversized Messages

Messages that are longer than 2048 characters (full length including address header, message-type code, message body, and \r terminator character) are formally defined to be "malformed" and should be handled the same as shorter malformed message strings as described above. Practically speaking, however, IMP-compliant applications should be written so as to be able to recognize such messages and signal them as being "oversized" so that the offending sending process can be debugged and the oversized messages eliminated.

A common practice is to allow input messages to be up to "`BUFSIZ`" characters long. `BUFSIZ` is the C/POSIX maximum size of the stream buffer defined in the `stdio.h` header file. In many Unix implementations (e.g., gcc) this is 8192 bytes, more than enough size to handle the maximum IMPv2 message size. The application can then accept well-formed but otherwise oversized message strings and complain appropriately so that the programmers can fix the problem.

6 Using IMPv2.5 in Interactive Applications

IMPv2.5 is designed to be human-readable. One of the principal design drivers behind this requirement was that it permits creation of an interactive command syntax that closely mirrors the full IMPv2 message syntax. This ability to craft "protocol level" communications by hand has greatly assisted engineering development and debugging of our instruments and associated software. This also allows a keyboard interface for an IMPv2 application to use the same command layer code as the non-interactive layers, which has simplifying coding interactive applications.

To enable this, several rules have emerged to make things easier for the user and the programmer.

6.1 Keyboard Commands

A command typed at an application's console can be thought of as an EXEC: message sent by an application to itself. For example, to print the current application status at the keyboard, one would type:

```
status
```

at the command-line prompt. This is equivalent to the full IMPv2 message string:

```
cam>cam EXEC: status\r
```

This is difficult and impractical to type, so for IMPv2 application command-line interfaces, the following rule applies:

Keyboard commands should be treated as implicitly self-addressed EXEC: commands.

This makes it straightforward to write a common command parser that handles internal and externally originating commands the same (greatly simplifying coding). Command interpreters do not need to distinguish between keyboard and other transports: they simply need consistent rules for handling EXEC: messages, and a way to "reply to self".

6.2 Outbound Messages

For outbound messages typed at the keyboard, this short-hand form should be used:

```
>FW filter 2\r
```

This example reads "Send to FW the command `filter 2`". The application's command interpreter needs only add the application's node address and then send it out via the network transport (serial, socket, etc. as required); everything else is to be sent as-is. This gives the command-line language a simple "Send-To" syntax enabled by the ">" character.

7 Examples

7.1 Software Handshaking

These are examples of software handshaking transactions using PING/PONG messages:

```
PR>IE PING\r
IE>PR PONG\r

IS>AL PING\r
IE>IS PONG\r
IC>IS PONG\r
PR>IS PONG\r
...
```

7.2 Command Requests

These are examples of valid command request messages:

```
PR>IE FILTER 1\r
PR>IE FILTER\r
PR>IE REQ: FILTER 1\r
```

Note that REQ: is *implicit* if no `msgType` string appears in the message.

These are examples of executive-override command request messages:

```
PR>CB EXEC: FSYNCH 0 DISK1 10\r
IS>TC EXEC: QUIT\r
```

7.3 Command-Completion and Command-Progress Messages

These are examples of valid command-completion messages:

```
IE>PR DONE: FILTER FILTPOS=1 FILTNAME='SDSS u'\r

CB>PR DONE: FSYNCH Read 10 images from DISK1 after forced synch\r
```

This is an example of a command transaction with progress STATUS: messages preceding a final DONE message:

```
PR>IE slitmask 4\r
IE>PR STATUS: slitmask Stowing SlitMask=2\r
IE>PR STATUS: slitmask Moving cassette to Slitmask=4\r
IE>PR STATUS: slitmask Inserting SlitMask=4 into beam\r
IE>PR DONE: slitmask SlitMask=4 SlitPos=Beam MaskID='A2218f12'\r
```

The first 3 messages report status of a verbose slitmask command, the final message signals successful completion along with relevant final state information coded as key=value pairs.