ICIMACS Messaging Protocol
Version 2
(IMPv2)

Document Number:	OSU-MODS-2003-007
Version:	1.1
Date:	2004 April 21
Prepared by:	R.W. Pogge & J.A. Mason, OSU

THE OHIO STATE UNIVERSITY

Distribution List		
Recipient	Institution/Company	Number of Copies
Richard Pogge Darren DePoy Jerry Mason	OSU OSU OSU	1 (file) 1 1

Document Change Record			
Version	Date	Changes	Remarks
0.1 0.2 1.0 1.1	2003-11-16 2003-12-16 2003-12-17 2004-04-21	Outline and working draft First distribution draft First Release Version Revisions and minor reorganization	Comments by Mason/Pogge

Contents

1 Introduction

1.1 Scope

This document describes IMPv2, version 2 of the ICIMACS Messaging Protocol (IMP) that is used by OSU instruments for interprocess communications. It extends and supercedes the original ICIMACS Messaging Protocol (IMPv1). 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. IMPv2 is thus a high-level protocol by this definition.

Except for the adoption of the standard ASCII character set, IMPv2 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).

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 Instrument Control and IMage ACquisition System, the generic name for the suite of 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 informational messages are passed between processes running in a "network" of ICIMACS hosts. Messages are formatted so as 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 main messaging protocol used in the data-taking systems of all OSU instruments deployed between 1995 and 2002.

The purpose of IMPv2 is to extend the original protocol to handle the more sophisticated new generation of instruments being built at OSU, and to formally define some 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 spectrograph.

The principal improvements of IMPv2 are as follows:

1. Expanded address names (from 2 to 8 characters) to permit greater flexibility in naming system nodes, and providing greater legibility of message address headers in large multi-node systems as we anticipate with complex instruments like MODS.
2. A new message type, EXEC:, which provides data-taking system applications with a way to lockout sensitive functions from remote execution unless specifically qualified as "executive override" command requests.
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 simple heartbeat mechanism for monitoring the health of client processes in complex systems.
6. Formal definition of rules for handling out-of-protocol messages.

IMPv2 is the messaging syntax used by the new ISIS data-taking system architecture. 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 formally 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. The MODS instruments for the LBT will be full IMPv2+ISIS systems.

3 ICIMACS Messaging Protocol version 2

3.1 Message Format

3.1.1 Basic Syntax

IMPv2 message strings take the following form:

```
src>dest Message_Type Message_Body\r
```

where:

`src`	node name of the sending process (8 chars max)
`dest`	node name of the destination process (8 chars max)
`Message_Type`	type of message being sent to `dest` (§3.3)
`Message_Body`	command or message text being sent to `dest` (§3.4).
`\r`	termination character ("carriage return" = ASCII 13)

Each of the main message string tokens are separated using the ASCII space character (ASCII 32). Multiple spaces are ignored. Within the `Message_Body`, spaces are used as token separators for arguments (either command arguments or parameter keyword=value pairs), 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.

3.1.2 Character Set

IMPv2 message strings must consist of only 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). The address header must be first component of a valid IMPv2 message. Extraneous leading spaces may be ignored, but no other characters may appear before an 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, 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 Special Broadcast Node Name "AL"

The node name "AL" is reserved as the broadcast address. An IMPv2-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 retains the 2-letter syntax for backwards compatibility with IMPv1.

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 the message text following is a(n) …
REQ:	command request [implicit if omitted]
EXEC:	command request with executive override
DONE:	command completion acknowledgment
STATUS:	informational status message
ERROR:	error message: the requested action terminated with errors
WARNING:	warning message: an unexpected result has occurred
FATAL:	an error condition that precludes acquiring or saving data

3.3.1 Command Request (REQ:)

REQ: is not new so much as now formally defined in IMPv2 as an explicit message type. 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 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: message type is new to IMPv2, and is designed to allow applications to define a subset of "expert" commands that are not executed ("locked out") if received as normal REQ: command requests. For example, a data-taking system application might be written so as to only accept a "QUIT" command (to terminate application execution) from that application's command-line interface, and 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.

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:)

The DONE: message type is the same as in IMPv1, and is used to signal to the recipient that a previous command request (implicit REQ: or EXEC:) has been successfully completed with no errors, and that no further messages regarding this command request will be generated. All "DONE:" messages are terminal in the sense that they complete a transaction, and do not require the receiving node 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 non-completion progress of a command.

Generally, 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 the same as in IMPv1, and is used to transmit status information to a recipient. When received in response to a command request (REQ: or EXEC:), it signals to the recipient that the command request is still in progress, and that further messages will be transmitted. 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 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 identical in both cases. Thus "DONE:" in this context is a "completion status" message, whereas "STATUS:" is an only to transmit intermediate command status (i.e., progress reports).

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 the same as defined in IMPv1. They are meant to signal error conditions of a range of severity have occurred. They may be sent in response to a command request, or 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 interpretations of the severity of a fault condition signified by each of the error message types are as follows:

A WARNING: message is to be sent when the requested action has been completed, but there were anomalies that need to be noted. A WARNING: message is the functional equivalent of

a STATUS: message, leaving it up to the recipient node to determine what additional actions to take in response. WARNING: class messages may also be broadcast to inform all nodes that some unusual condition, possibly requiring attention, has occurred. If a warning is generated during the course of command execution that otherwise completes, a subsequent DONE: message will still be required in order to ensure that the command request transaction correctly terminates. Similarly, if after the warning the command request aborts with errors, an ERROR: or FATAL: message (as appropriate) is required to terminate the transaction.

An ERROR: message is to be sent when a requested action results in an error that prevents it from being carried out to completion. These conditions include any condition which causes abnormal termination of a request. Specific instances include command syntax errors (e.g., the action requested or its command arguments are unknown to the sending node), invalid command arguments (e.g., a requested parameter setting is out of the allowed range), and the command request was valid and attempted, but a fault occurred which aborted the request before completion (e.g., a mechanism was requested to move but did not respond within some reasonable timeout interval). ERROR: messages should only be sent when the fault condition is of sufficient severity that the command request could not be completed; see WARNING: for when the command request could (apparently) be completed but anomalies occurred.

A FATAL: error message is a special condition in which an error has occurred of sufficient severity that physical intervention is required to correct the fault condition. Further, nodes receiving a FATAL: class error message, if so configured, should abort whatever procedures they are executing and enter a "safe mode" awaiting further instructions. 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 Message Body

The `Message_Body` follows the message type code, and contains the essential information of the message.

Command requests are usually posed in "command arg1 arg2 …" syntax familiar from most command-line interpreters. The syntax should be appropriate for the command-line syntax of the receiving process.

Command-completion (DONE:), informational status (STATUS:), and error messages may consist of unstructured but otherwise human readable text (e.g., "ERROR: Requested filter 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.

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 separators between adjacent key=value pairs.

3.4.1 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.

3.4.2 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.

3.4.3 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 IMPv2 multi-word string representations. The () delimiters are from IMPv1, and retained for backwards compatibility. Parsers in IMPv2-compliant applications need to be flexible enough to handle both instances of multi-word strings.

3.4.4 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".

3.5 Message Termination

All messages must be terminated by a "carriage return" character (CR, Ctrl+M, ASCII 13), or "`\r`" in common POSIX 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 § 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).

4.1 Software Communications Handshaking: PING/PONG

To provide for software handshaking between two nodes, the PING and PONG messages are used. PING/PONG are retained as-is from IMPv1.

PING is an "I am here" message, used to initiate software handshaking between nodes. For example, on startup a client node would "PING" the message-passing server node to inform it that it is present and introduce itself. PINGs may be broadcast into a message-passing network to inform all active nodes of the emergence of a new node.

PONG acknowledges a PING and completing 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 nodes may ignore any additional information appearing after the PONG.

4.2 Process Heartbeat Messages

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

```
tcs>isis\r
```

would be used by the client node named "tcs" to inform a server node "isis" that it is still online. The heartbeat must be properly terminated like all IMPv2 messages (§3.5).

Such a bare header would generate a messaging syntax error under IMPv1. All IMPv2 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 Messages

Any message shall be considered "out-of-protocol" (O-o-P) if it violates any of the syntax requirements described in §3. This section describes how IMPv2 system nodes should respond to receipt of O-o-P messages to prevent problems.

5.1 Malformed Messages

Malformed messages are those that have most of the components of a valid IMPv2 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.

IMPv2-compliant applications should be written so as 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.

IMPv2-compliant applications may elect to treat \n as \r (i.e., implicit translation), much as is done, for example, by the standard Unix 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 standard IMPv2 syntax. For example, ASCII text received on a serial interface in which the other end of the serial line is not connected to an IMPv2-compliant application (e.g., an application accidentally opens up a port it shouldn't have). In these cases no exceptions are to be made. The messages are to be treated as "extraneous" and ignored by the IMPv2-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 IMPv2-compliant messages is required, an appropriate "filter application" should be written that buffers communications with the non-compliant process and translates it into IMPv2-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

IMPv2 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, IMPv2-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 used in many of our IMPv2-compliant Unix applications 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 in Interactive Applications

IMPv2 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, a number of 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 IMPv2 application to itself. For example, to print the current application status at the keyboard, one would type:

```
status
```

at the command-line prompt. This can be seen as equivalent to the full IMPv2 message string:

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

This is difficult to type, so we have made the following design principle for IMPv2 application command-line interfaces:

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

This makes it very simple 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 is used:

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

This example reads "Send to node FW the command `filter 2`". The applications'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.