pvAccess Protocol Specification

EPICS v4 Working Group, Fourth (in progress) Public Working Draft, 16-October-2015

This version:
pvAccess_Protocol_Specification_20151016.html
Latest version:
pvAccess_Protocol_Specification.html
Previous version:
pvAccess_Protocol_Specification_20140407.html (3rd Public Working Draft)

Editors:
Matej Sekoranja, Cosylab
Marty Kraimer, BNL
Greg White, SLAC, PSI
Andrew Johnson, APS (Invited Expert)
Benjamin Franksen, HZB (Invited Expert)
Michael Abbott, DLS (Invited Expert)
Philip Duval, DESY (Invited Expert)

Abstract

This document defines the EPICS communication protocol called "pvAccess". pvAccess is a high-performance network communication protocol for signal monitoring and scientific data services interconnect. It is designed to support the structured data types of the EPICS 7 (and above) "shared memory" data exchange system called pvData, for optimized interoperability of control system endpoints. It is a successor of EPICS Channel Access.

The connection setup requirements and individual message constructs of pvAccess are described. It is intended that sufficient detail is given for a reader to create an interoperable pvAccess implementation. The protocol and a reference implementation have been created by the EPICS Version 4 Working Group.

EPICS is a computer platform for building the control systems of large scientific instruments. For more information about EPICS, please refer to the home page of the Experimental Physics and Industrial Control System.

Status of this Document

This is the 16 October 2015 version of the pvAccess Protocol Specification. This version constitutes the first publication of the Third Public Working Draft. The Third Public Working Draft includes in particular support for unsigned integer, unions, and material and revisions from the Invited Experts above. The Public Working Drafts are intended for the EPICS community to review and comment. Resulting comments will drive subsequent revisions of the specification and the EPICS Version 4 Working Group's reference implementation.

Missing aspects of this specification are described in the last section at the end of this document. It is expected that the next draft will address these remaining items as a Last Call draft, and be followed by publication of the specification.

The present implementation of pvAccess largely reflects the specification as written here. Another document will soon be written to track the status of the reference implementation with respect to the specification, showing what has yet to be implemented.

The terms MUST, MUST NOT, SHOULD, SHOULD NOT, REQUIRED, and MAY when highlighted (through style sheets, and in uppercase in the source) are used in accordance with RFC 2119 [RFC2119]. The term NOT REQUIRED (not defined in RFC 2119) indicates exemption.

In general the text in this document is intended to be "normative", which is to say it constitutes a formal specification of protocol itself. As such that text is concise, algorithmic, and describes the protocol systematically. Such text is in the default font of the document. The functional consequences of the specification so described, although not normative, may be important. Such non-normative text is in *italics*.

Table of Contents

Overview

pvAccess is a high-performance network communication protocol. It is primarily designed for efficient signal monitoring and the data requirements of a service oriented architecture.

pvAccess is a successor of EPICS Channel Access bib:caref. It is the standard protocol of EPICS Version 4 (V4), EPICS 7 and above when both endpoints of a channel are EPICS V4 agents. pvAccess can also connect to EPICS V3 IOC Process Databases using the EPICS 7 QSRV module.

TCP/IP is used for data transmission. UDP/IP is normally used for discovery, although discovery over TCP/IP is also allowed. The discovery mechanism allows the use of other implementations (e.g. UDP/IP for data transmission). The protocol itself supports IPv6, i.e. all addresses are IPv6 encoded.

Port numbers 5075 (tcp connection port, accepted by IANA) and 5076 (udp broadcast port) are used by default. These default connection ports SHOULD be used if free, otherwise a dynamically allocated port SHOULD be used as a fallback. pvAccess implementations SHOULD allow alternative default connection ports to be configured.

To support multiple local sockets at port 5076 to able all to receive unicast messages over the UDP a multicast group on local network interface at address 224.0.0.128, port 5076, is used. Any UDP message flagged as unicast received at port 5076 MUST be forwarded to the multicast group with unicast flag cleared.

pvAccess was designed to support pvData bib:pvdatarefcpp, bib:pvdatarefjava. pvData is the control data interface of EPICS V4 endpoints, such as user agent software and EPICS V4 IOCs. Together with pvAccess, they support essentially a client-server shared memory system optimized for efficiency (optimal zero-copy etc) and control (PV locking, alarms etc). The protocol aims to send the minimum number of bits necessary to inform peers of changes in endpoint data values subject to performance considerations. That is, it combines CPU and wire data size considerations to optimize overall control network throughput. pvAccess supports segmented messages and thus allows the sending of large amounts of data using optimal buffer sizing. The maximum message size is not limited with respect to the send or receive buffer sizes. *In practice, this means there is no need for a pvAccess equivalent of Channel Access' EPICS_CA_MAX_ARRAY_BYTES.*

The pvAccess protocol definition consists of three major parts:

- A set of data encoding rules that determine how the various data types are encoded and deserialized
- A set of rules that determine how client and server agree on a particular encoding
- A number of message types, that define the interchange between endpoints, together with rules which specify what message is to be sent under what circumstances.

Data Encoding

The goals of pvAccess data encoding are simplicity and efficiency. In keeping with these goals, the encoding does not align primitive types on word boundaries and therefore eliminates the wasted space and additional complexity that alignment requires. pvAccess data encoding simply produces a stream of contiguous bytes; in general message data does not contain padding bytes and an implementation MUST NOT try to align data on word boundaries.

For connection-oriented communication (TCP/IP), the server MUST notify the client what byte order to use. Each message contains an endianness flag in order to allow all the intermediates to forward data without requiring it to be unmarshaled (so that the intermediates can forward requests by simply copying blocks of binary data) and in order not to require a specific byte order for connection-less protocols (UDP/IP).

For clarity, this document separates user data (consisting of basic types, string, arrays, structures) from meta data, which exists only at the protocol level (Status, BitSet). Meta data can be of user data type, but not the other way around.

Sizes

Many of the types involved in the data encoding, as well as several protocol message components, have an associated size (or "count"). Size values MUST always be a non-negative integer and encoded as follows:

1. If the number of elements is less than 255, the size MUST be encoded as a single byte containing an unsigned 8-bit integer indicating the number of elements
2. If the number of elements is less than 2^31-1, then the size MUST be encoded as an unsigned 8-bit integer with value 255, followed by a positive signed 32-bit integer indicating the number of elements
3. If the number of elements is greater than or equal to 2^31-1, then the size MUST be encoded as an unsigned 8-bit integer with value 255, followed by a positive signed 32-bit integer with value 2^31-1, followed by a positive signed 64-bit integer indicating the number of elements. This implies a maximum size of 2^63-1.

Using this encoding to indicate size is significantly cheaper than always using a 32-bit (or even 64-bit) integer to store the size. This is especially true when marshalling sequences of short strings; counts of up to 254 require only a single byte instead of four. This comes at the expense of counts greater than 254, which require five bytes instead of four. However, for sequences or strings of length greater than 254, the extra byte is insignificant.

User Data

Basic Types

The basic types MUST be encoded as shown in the Table 1. Signed integer types (byte, short, int, long) MUST be represented as two's complement numbers. Floating point types (float, double) MUST use the IEEE-754 standard formats bib:ieee754wiki.

Type	Encoding
boolean	A single byte with value non-zero value for true, zero for false.
byte	Signed 8-bit integer.
ubyte	Unsigned 8-bit integer.
short	Signed 16-bit integer.
ushort	Unsigned 16-bit integer.
int	Signed 32-bit integer.
uint	Unsigned 32-bit integer.
long	Signed 64-bit integer.
ulong	Unsigned 64-bit integer.
float	32-bit float (IEEE-754 single-precision float).
double	64-bit float (IEEE-754 double-precision float).

Encoding for basic types.

Note on boolean encoding: a receiver MUST NOT assume that a boolean value of true is represented by any special non-zero number, nor that the same sender consistently uses the same number.

Arrays

Variable-size Arrays

Variable-size arrays MUST be encoded as a size representing the number of elements in the array, followed by the elements encoded as specified for their type (as specified in these sections).

Bounded-size Arrays

Bounded-size arrays MUST be encoded as a size representing the number of elements in the array, followed by the elements encoded as specified for their type (as specified in these sections). The size MUST be less then or equal to the array's declared bound.

Fixed-size Arrays

Fixed-size arrays MUST be encoded as elements encoded as specified for their type (as specified in these sections). The number of elements encoded MUST equal to the array's fixed size.

Strings

Strings are encoded as arrays of bytes. The actual content (the bytes in the array) MUST be a valid UTF-8 encoded string.

Particularly, this means that strings MUST be encoded as a size, followed by the string contents in a UTF-8 format as bytes. Size gives the number of bytes that follow it and not the number of UTF-8 characters. UTF-8 multi-byte characters MUST NOT be broken. An empty string MUST be encoded with a size of zero.

Implementations that internally use a zero byte or a zero character to indicate end-of-string SHOULD NOT include a terminating zero byte in the pvAccess string encoding. 'null' strings are not supported.

On the wire, pvAccess MUST transmit all strings as Unicode strings in UTF-8. Non-C++ bindings of the implementations SHOULD use strings in their language-native Unicode representation and convert automatically to and from UTF-8 for transmission, so applications can transparently use characters from non-English alphabets. However, for C++, how strings are represented inside a process depends on the platform as well as the mapping that is chosen for a particular string. The default mapping to use is std::string.

Bounded Strings

Same as strings, just that size MUST be less than or equal to the string's bound.

Structures

Structures MUST be encoded by appending the data of all comprising fields in the order in which the fields have been defined. A structure can contain a structure and an union (see below) for its field.

Unions

Unions MUST be encoded as a selector value (encoded as a size), followed by the selected union member data. The selector chooses one member of a union as specified in the union introspection data, so must be a value in the range 0..N-1 where N is the number of union members. A union can contain a structure and a union for its field.

Variant Unions

Variant Unions are open ended union type, also known as *any* type. Variant Unions MUST be encoded as a introspection data (*Field*) description of the encoded value, followed by the encoded value itself.

Encoding Example

Given the following structure:

```
structure
    byte[] value [1,2,3]
    byte<16> boundedSizeArray [4,5,6,7,8]
    byte[4] fixedSizeArray [9,10,11,12]
    structure timeStamp
        long secondsPastEpoch 0x1122334455667788
        int nanoSeconds 0xAABBCCDD
        int userTag 0xEEEEEEEE
    structure alarm
        int severity 0x11111111
        int status 0x22222222
        string message Allo, Allo!
    union valueUnion
        int  0x33333333
    any variantUnion
        string  String inside variant union.
```

The above would be serialized as illustrated below (when using big-endian byte order, *valueUnion* selector with value 1 is selected):

```
Hexdump [Serialized structure] size = 85
03 01 02 03  05 04 05 06  07 08 09 0A  0B 0C 11 22  .... .... .... ..."
33 44 55 66  77 88 AA BB  CC DD EE EE  EE EE 11 11  3DUf w... .... ....
11 11 22 22  22 22 0B 41  6C 6C 6F 2C  20 41 6C 6C  ..""  "".A llo,  All
6F 21 01 33  33 33 33 60  1C 53 74 72  69 6E 67 20  o!.3 333` .Str ing
69 6E 73 69  64 65 20 76  61 72 69 61  6E 74 20 75  insi de v aria nt u
6E 69 6F 6E  2E                                     nion .
```

Meta Data

BitSets

BitSet is a data type that represents a finite sequence of bits.

BitSet is encoded as a byte array. Bits are serialized in groups of eight in ascending order (LSB to MSB). Serialization is size optimized to send only the least possible number of bytes that encode all the bits in the set.

Examples of BitSet serialization:

```
Hexdump [{}] size = 1
00                                                 .

Hexdump [{0}] size = 2
01 01                                              ..

Hexdump [{1}] size = 2
01 02                                              ..

Hexdump [{7}] size = 2
01 80                                              ..

Hexdump [{8}] size = 3
02 00 01                                           ...

Hexdump [{15}] size = 3
02 00 80                                           ...

Hexdump [{55}] size = 8
07 00 00 00  00 00 00 80                            .... ....

Hexdump [{56}] size = 9
08 00 00 00  00 00 00 00  01                       .... .... .

Hexdump [{63}] size = 9
08 00 00 00  00 00 00 00  80                       .... .... .

Hexdump [{64}] size = 10
09 00 00 00  00 00 00 00  00 01                    .... .... ..

Hexdump [{65}] size = 10
09 00 00 00  00 00 00 00  00 02                    .... .... ..

Hexdump [{0, 1, 2, 4}] size = 2
01 17                                              ..

Hexdump [{0, 1, 2, 4, 8}] size = 3
02 17 01                                           ...

Hexdump [{8, 17, 24, 25, 34, 40, 42, 49, 50}] size = 8
07 00 01 02  03 04 05 06                            .... ....

Hexdump [{8, 17, 24, 25, 34, 40, 42, 49, 50, 56, 57, 58}] size = 9
08 00 01 02  03 04 05 06  07                       .... .... .

Hexdump [{8, 17, 24, 25, 34, 40, 42, 49, 50, 56, 57, 58, 67}] size = 10
09 00 01 02  03 04 05 06  07 08                    .... .... ..

Hexdump [{8, 17, 24, 25, 34, 40, 42, 49, 50, 56, 57, 58, 67, 72, 75}]
size = 11
0A 00 01 02  03 04 05 06  07 08 09                 .... .... ...
```

```
Hexdump [{8, 17, 24, 25, 34, 40, 42, 49, 50, 56, 57, 58, 67, 72, 75,
81, 83}] size = 12
0B 00 01 02  03 04 05 06  07 08 09 0A               .... .... ....
```

Partial Structure Serialization

Each structure can (depending on message definition) have a BitSet instance defining what subset of that structure's fields have been serialized. This allows partial serialization of structures. That is, serializing only fields that have changed rather than the entire structure. Each node of a structure corresponds to one bit; if a bit is set then its corresponding field has been serialized, otherwise not. BitSet does not apply to array elements.

This example shows how bits of a BitSet are assigned to the fields of a structure:

```
bit#    field
0    structure
1        structure timeStamp
2            long secondsPastEpoch
3            int nanoSeconds
4            int userTag
5        structure[] value
           structure org.epics.ioc.test.testStructure
               double value
               structure location
                   double x
                   double y
           structure org.epics.ioc.test.testStructure
               double value
               structure location
                   double x
                   double y
6        string factoryRPC
7        structure arguments
8            int size
```

The structure above requires a BitSet that contains 9 bits. If the bit corresponding to a structure node is set, then all the fields of that node MUST be serialized.

Status

pvAccess defines a structure to inform endpoints about completion status. It is nominally defined as:

```
struct Status {
    byte type;      // enum { OK = 0, WARNING = 1, ERROR = 2, FATAL = 3
}
    string message;
    string callTree;    // optional (provides more context data about
the error), can be empty
};
```

In practice, since the majority of Status instances would be OK with no message and no callTree, a special definition of Status SHOULD be used in the common case that all three of these conditions are met; if Status is OK and no message and no callTree would be sent, then the special type value of `-1` MAY be used, and in this case the string fields are omitted:

```
struct StatusOK {
    byte type = -1;
};
```

Examples of Status serialization:

```
Hexdump [Status OK] size = 1
FF                                                  .

Hexdump [WARNING, "Low memory", ""] size = 13
01 0A 4C 6F  77 20 6D 65  6D 6F 72 79  00           ..Lo w me mory .

Hexdump [ERROR, "Failed to get, due to unexpected exception", (call
tree)] size = 264
02 2A 46 61  69 6C 65 64  20 74 6F 20  67 65 74 2C  .*Fa iled  to  get,
20 64 75 65  20 74 6F 20  75 6E 65 78  70 65 63 74   due  to  unex pect
65 64 20 65  78 63 65 70  74 69 6F 6E  DB 6A 61 76  ed e xcep tion .jav
61 2E 6C 61  6E 67 2E 52  75 6E 74 69  6D 65 45 78  a.la ng.R unti meEx
63 65 70 74  69 6F 6E 0A  09 61 74 20  6F 72 67 2E  cept ion. .at  org.
65 70 69 63  73 2E 63 61  2E 63 6C 69  65 6E 74 2E  epic s.ca .cli ent.
65 78 61 6D  70 6C 65 2E  53 65 72 69  61 6C 69 7A  exam ple. Seri aliz
61 74 69 6F  6E 45 78 61  6D 70 6C 65  73 2E 73 74  atio nExa mple s.st
61 74 75 73  45 78 61 6D  70 6C 65 73  28 53 65 72  atus Exam ples (Ser
69 61 6C 69  7A 61 74 69  6F 6E 45 78  61 6D 70 6C  iali zati onEx ampl
65 73 2E 6A  61 76 61 3A  31 31 38 29  0A 09 61 74  es.j ava: 118) ..at
20 6F 72 67  2E 65 70 69  63 73 2E 63  61 2E 63 6C   org .epi cs.c a.cl
69 65 6E 74  2E 65 78 61  6D 70 6C 65  2E 53 65 72  ient .exa mple .Ser
69 61 6C 69  7A 61 74 69  6F 6E 45 78  61 6D 70 6C  iali zati onEx ampl
65 73 2E 6D  61 69 6E 28  53 65 72 69  61 6C 69 7A  es.m ain( Seri aliz
61 74 69 6F  6E 45 78 61  6D 70 6C 65  73 2E 6A 61  atio nExa mple s.ja
76 61 3A 31  32 36 29 0A                            va:1 26).
```

Introspection Data

Introspection data describes the type of a user data item. It is not itself user data, but rather meta data. Introspection data appears in one of four forms: no introspection data (NULL_TYPE_CODE), a full type description (FULL_TYPE_CODE), a type identifier (ONLY_ID_TYPE_CODE), both (FULL_WITH_ID_TYPE_CODE), according to the table "Encoding of Introspection Data".

The sender MUST send introspection data, but is free to chose one of the above methods. Sending FULL_WITH_ID_TYPE_CODE defines the type identifier for subsequent sends using ONLY_ID_TYPE_CODE. Therefore, before sending ONLY_ID_TYPE_CODE, the sender MUST have previously sent at least one FULL_WITH_ID_TYPE_CODE with the same type identifier to the same receiver.

Since user data types can be arbitrarily complex, introspection data SHOULD be sent only once per type and receiver combination. The mapping of dynamically assigned type identifier (ID) to introspection data MUST be cached on the receiver side, and SHOULD be cached and re-used on the sender side. ID MUST be encoded as short and MUST be valid only within one connection. Moreover, IDs MUST be assigned only by the sender. The receiver MUST keep track of the IDs and use them to identify deserializations. Since communication is full-duplex this implies there MUST be two introspection registries per connection. The sender MAY override a previously assigned ID by simply assigning the ID to a new introspection data instance. The introspection registry size MUST be negotiated when each connection is established.

Field Encoding	Name	Description
0xFF	**NULL_TYPE_CODE**	No introspection data (also implies no data).
0xFE + ID	**ONLY_ID_TYPE_CODE**	Serialization contains only an ID (that was assigned by one of the previous FULL_WITH_ID_TYPE_CODE or FULL_TAGGED_ID_TYPE_CODE descriptions).
0xFD + ID + FieldDesc	**FULL_WITH_ID_TYPE_CODE**	Serialization contains an ID (that can be used later, if cached) and full interface description. Any existing definition with the same ID is overriden.
0xFC + ID + tag + FieldDesc	**FULL_TAGGED_ID_TYPE_CODE**	Serialization contains an ID (that can be used later, if cached), tag (of integer type) and full interface description. Any existing definition with the same ID is overriden. A tag must guarantee that the same (ID, FieldDesc) pair has the same tag and any previous definition with the same ID and different FieldDesc has a different tag. This identifies whether the definition with given ID overrides already existing one and allow receivers to skip deserialization of FieldDesc, if tags

		match. SHOULD be used in non-reliable transport systems only.
0xFB - 0xE0	RESERVED	Reserved for future usage, MUST NOT be used.
FieldDesc (0xDF - 0x00)	**FULL_TYPE_CODE**	Serialization contains only full interface description.

Encoding of Introspection Data (called Field for future reference).

Each instance of a Field introspection description (FieldDesc) MUST be encoded as a byte that consists of 2 nibbles (4-bits). The upper nibble (Most Significant Bits, MSBs) is used for the type selector and flags. The lower nibble (bits 7-5) is type dependent and used for size encoding.

bit	Value	Description
7-5	111	reserved (MUST never be used)
	110	reserved (MUST not be used)
	101	
	100	complex
	011	**string**
	010	floating-point
	001	integer
	000	**boolean**
3-4	11	fixed-size array flag
	10	bounded-size array flag
	01	variable-size array flag
	00	scalar flag
2-0		type (bits 7-5) depended

Type Encoding.

bit	Value	Type Name
2	1	unsigned flag
	0	signed flag
1-0	11	**long**
	10	**int**
	01	**short**
	00	**byte**

Integer Type Size Encoding (type = '0b001').

bit	Value	Type Name	IEEE 754-2008 Name
2-0	111	reserved	
	110		
	101		
	100	reserved	binary128 (Quadruple)
	011	**double**	binary64 (Double)
	010	**float**	binary32 (Single)
	001	reserved	binary16 (Half)
	000	reserved	

Floating-Point Size Encoding (type = '0b010').

bit	Value	Type Name
2-0	111	reserved
	110	
	101	
	100	
	011	**bounded string**
	010	**variant union**
	001	**union**
	000	**structure**

Complex Type Encoding (type = '0b100').

FieldDesc Encoding	Description
`0bxxx00xxx`	Scalar.
`0bxxx01xxx`	Variable-size array of scalars.
`0bxxx10xxx` + bound (encoded as size)	Bounded-size array of scalars.
`0bxxx11xxx` + fixed size (encoded as size)	Fixed-size array of scalars.
`0b10000000` + identification string + (field name, FieldDesc)[]	Structure.
`0b10001000` + structure FieldDesc	Array of structures.
`0b10000001` + identification string + (field name, FieldDesc)[]	Union.
`0b10001001` + union FieldDesc	Array of unions.
`0b10000010`	Variant union.
`0b10001010`	Array of variant unions.
`0b10000110` + bound (encoded as size)	Bounded string.

FieldDesc Encoding.

For all other types, bits 2-0 MUST be '0b0000'.

Structure, union, and bounded string (and all their arrays) REQUIRE more description. A structure REQUIRES its identification string and a named array of Fields - size followed by one or more (field name, FieldDesc) pairs. Arrays of structures/unions REQUIRE an introspection data of a structure/union defining an array element type.

Example #1

Given the following structure, as may be expressed by a pvData Structure:

```
timeStamp_t
    long secondsPastEpoch
    int nanoSeconds
    int userTag
```

The introspection description of the above structure is be encoded by pvAccess as the following:

```
Hexdump [Serialized structure IF] size = 57
FD 00 01 80  0B 74 69 6D  65 53 74 61  6D 70 5F 74  .... .tim eSta mp_t
03 10 73 65  63 6F 6E 64  73 50 61 73  74 45 70 6F  ..se cond sPas tEpo
63 68 23 0B  6E 61 6E 6F  53 65 63 6F  6E 64 73 22  ch#. nano Seco nds"
07 75 73 65  72 54 61 67  22                        .use rTag "
```

Example #2

Given the following structure, as may be expressed by a pvData Structure:

```
exampleStructure
    byte[] value
    byte<16> boundedSizeArray
    byte[4] fixedSizeArray
    time_t timeStamp
        long secondsPastEpoch
        int nanoseconds
        int userTag
    alarm_t alarm
        int severity
        int status
        string message
    union valueUnion
        string stringValue
        int intValue
        double doubleValue
    any variantUnion
```

The introspection description of the above structure would be encoded by pvAccess as the following:

```
Hexdump [Serialized structure IF] size = 243
FD 00 01 80  10 65 78 61  6D 70 6C 65  53 74 72 75  .... .exa mple Stru
63 74 75 72  65 07 05 76  61 6C 75 65  28 10 62 6F  ctur e..v alue (.bo
75 6E 64 65  64 53 69 7A  65 41 72 72  61 79 30 10  unde dSiz eArr ay0.
0E 66 69 78  65 64 53 69  7A 65 41 72  72 61 79 38  .fix edSi zeAr ray8
04 09 74 69  6D 65 53 74  61 6D 70 FD  00 02 80 06  ..ti meSt amp. ....
74 69 6D 65  5F 74 03 10  73 65 63 6F  6E 64 73 50  time _t.. seco ndsP
61 73 74 45  70 6F 63 68  23 0B 6E 61  6E 6F 73 65  astE poch #.na nose
63 6F 6E 64  73 22 07 75  73 65 72 54  61 67 22 05  cond s".u serT ag".
61 6C 61 72  6D FD 00 03  80 07 61 6C  61 72 6D 5F  alar m... ..al arm_
74 03 08 73  65 76 65 72  69 74 79 22  06 73 74 61  t..s ever ity" .sta
74 75 73 22  07 6D 65 73  73 61 67 65  60 0A 76 61  tus" .mes sage `.va
6C 75 65 55  6E 69 6F 6E  FD 00 04 81  00 03 0B 73  lueU nion .... ...s
74 72 69 6E  67 56 61 6C  75 65 60 08  69 6E 74 56  trin gVal ue`. intV
61 6C 75 65  22 0B 64 6F  75 62 6C 65  56 61 6C 75  alue ".do uble Valu
65 43 0C 76  61 72 69 61  6E 74 55 6E  69 6F 6E FD  eC.v aria ntUn ion.
00 05 82                                            ...
```

Connection Management

pvAccess uses the concept of a "channel" to denote a connection to a single named resource that resides on some server. Channels are subordinate to the TCP connection between a client and server: a channel can only be created if a TCP connection has already been established; likewise, if the TCP connection is terminated, then all subordinate channels are implicitly destroyed.

Each TCP connection has associated Quality of Service (QoS) parameters. Regardless of how many channels are handled by either client or server, each client and server pair MUST be connected with exactly one TCP connection for each QoS parameter value.

When establishing a TCP connection, a simple handshake MUST be performed. The client opens a TCP connection to the server and waits until the Connection Validation message is received. The server MUST initially send a Set byte order control message to notify the client about the byte order to be used for this TCP connection. After that the server MUST send the Connection Validation message. If the client correctly decodes messages it MUST respond with a Connection Validation response message. Now the connection is verified and the client may start sending requests. The client SHOULD keep the connection established until the last active channel gets destroyed. However, to optimize resource reallocation it MAY delay connection destruction.

Both parties MUST constantly monitor whether the connection is valid and not simply rely on TCP mechanisms. pvAccess achieves this by sending some small amount of data with a minimum period. If there is no send operation otherwise called within a predetermined period of time (SHOULD be 15 seconds), an echo message MUST be sent. In case of connection failure, TCP will report a connection loss on send. If there is no response in a predetermined period of time, the connection SHOULD be marked as unresponsive. An echo message MUST be periodically sent until a response is received or the connection is reported to be lost. If an echo response is received and transport is marked as unresponsive, then transport SHOUD be reported to be responsive.

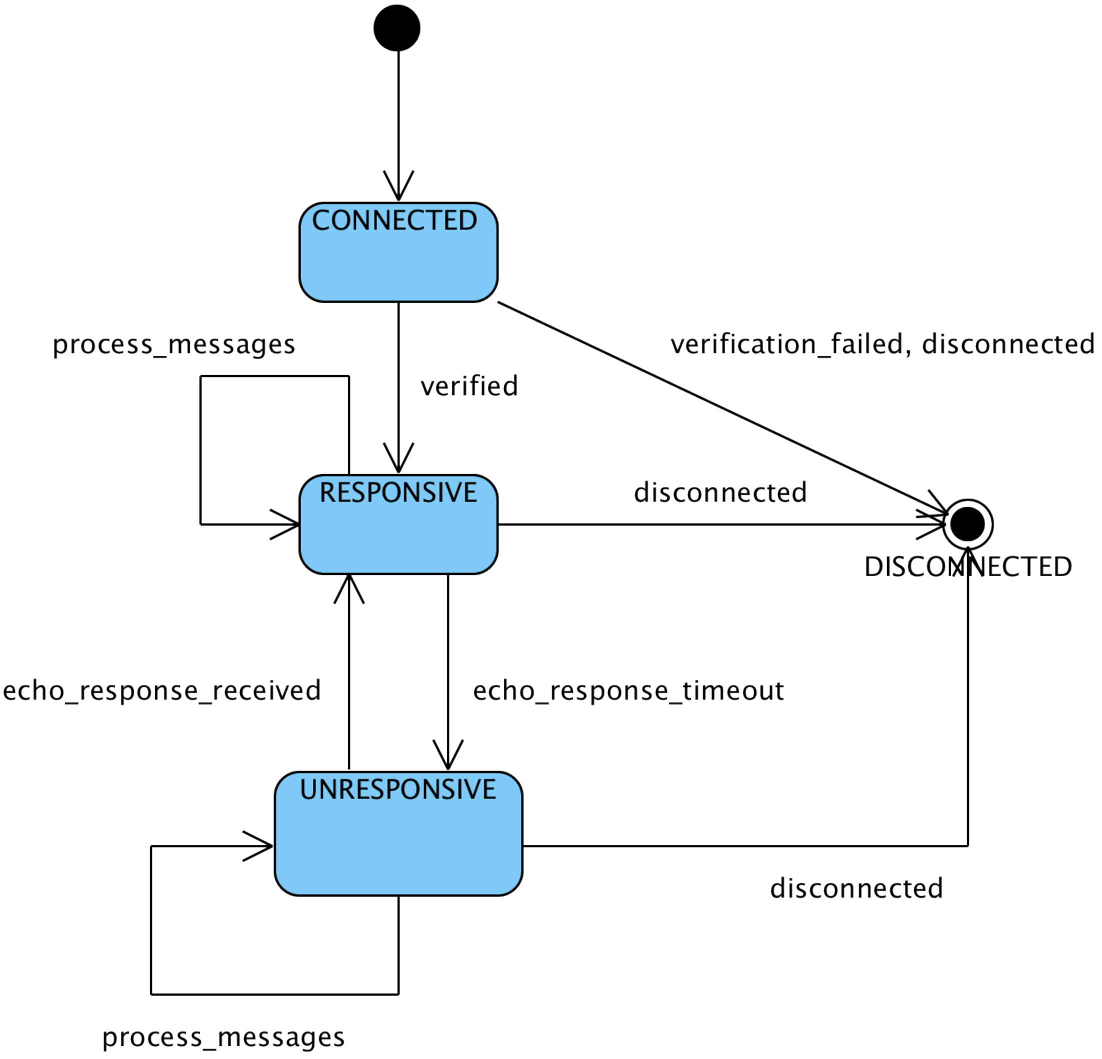

Figure 1 Connection State Diagram.

When connection is terminated all related resources MUST be freed. On the server side all channels including their requests MUST be destroyed (this includes all *serverChannelIDs*). On the client side all channels and their requests MUST be put to disconnected state and searching for channels initiated. *clientChannelIDs* and *requestIDs* SHOULD be retained until channel or request are destroyed on client side. Once IDs are freed they MAY be recycled - used for other channels/requests in the future.

When disconnected client channels are found on the network and connection is re-established, channels are put back to connected state and all their requests re-initialized; in addition, monitors are re-started.

Channel Life-cycle

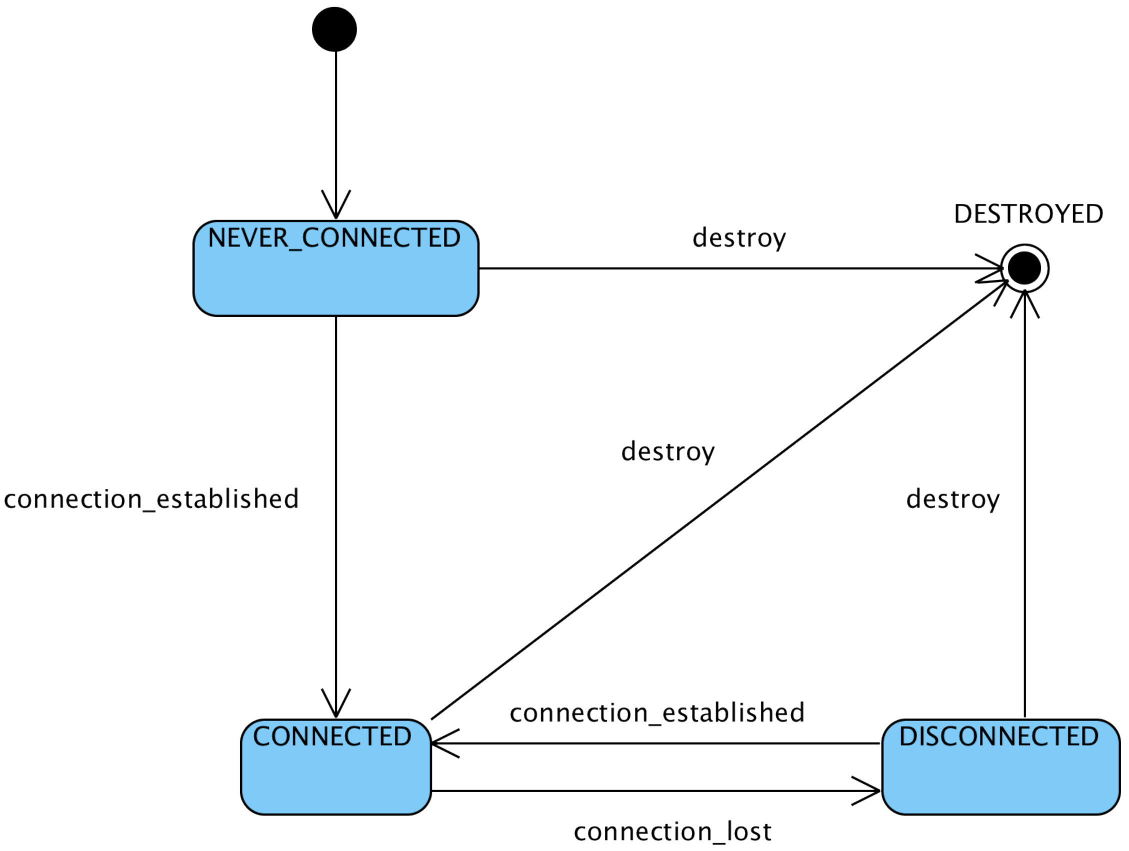

Figure 2 Channel State Diagram.

When a channel is instantiated by a client application, its state MUST be set to a NEVER_CONNECTED state. This indicates that the channel is currently being connected for the first time. The connection proccess within the client MUST repetedly attempt to find a server hosting the channel by broadcasting or multicasting channel search requests. When a server response is received, the client MUST connect to the server responding to the search request using the protocol and address data from the search request response. If a connection has already been established by the client, it MUST be reused. A client API MAY also allow a user-specified server address; in this case, the searching process would be bypassed and the specified server address data used directly.

When a connection is established and verified, a channel create request message MUST be sent by the server. When the client receives a channel create response message with a success status, it MUST set the channel to the CONNECTED state.

A channel MUST be in a CONNECTED state to be able to accept channel related requests.

When the connection is lost, the channel state MUST be set to DISCONNECTED. In this state, clients MUST start the connection process as described above. On reconnect, the channel's state MUST be set back to CONNECTED.

A channel MAY be destroyed any time (in any state) and then its state MUST be set to DESTROYED. Once the channel is destroyed, it MUST NOT be used anymore.

Channel Request Life-cycle

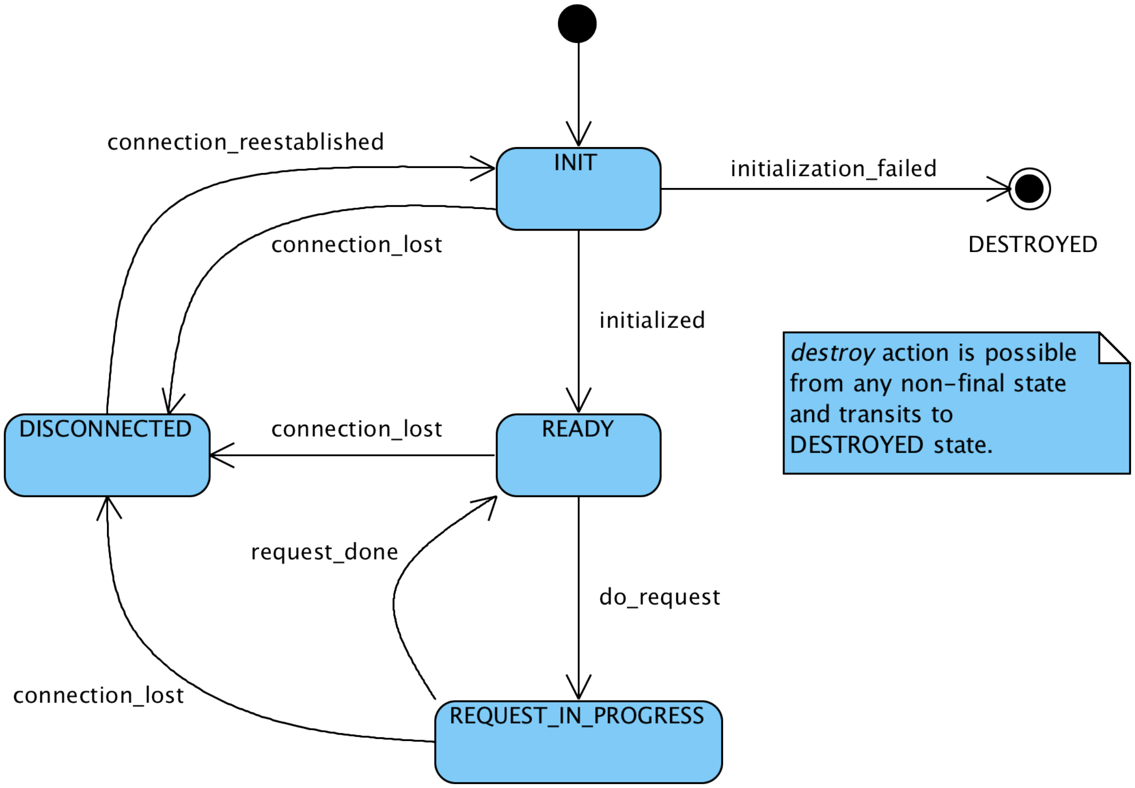

Figure 3. Channel Request State Diagram.

Channel requests (get, put, get-put, RPC, process) have a state. When instantiated, they MUST be set to the INIT state. A specific per request initialization message MUST be sent to the server. The request MUST NOT be used until a successful initialization response is received from the server and put to the READY state. If initialization fails, the client MUST be notified about the failure and the request put to the DESTROYED state.

Actual actions, e.g. get, MAY only be invoked when a request is in the READY state. When one action is in progress, the request is put into the REQUEST_IN_PROGRESS state and set back to the READY state when the action is completed. This implies that actions MUST NOT be run in parallel.

When a connection is lost, a request MUST be put into the DISCONNECTED state and automatically reinitialized when the connection is reestablished (as if the request were newly instantiated).

A pending request MAY be canceled. Actual cancellation MAY be ignored, however completion of the request MUST be always reported via request completion callback mechanism.

A request MAY be destroyed at any time (in any state) and then its state MUST be set to DESTROYED. Once the request is destroyed, it MUST NOT be used anymore.

Flow Control

This section is **not** intended to be normative. It is given only to help developers write agents that implement pvAccess optimally with respect to monitoring. This section does not describe the protocol itself.

A pvAccess implementation SHOULD implement flow control such that each endpoint should try to send as much monitoring data as it can subject to an upper limit calculated with respect to the amount of the other party's free receive buffer size. Were this limit to be reached, monitors would start piling up in the monitors' circular buffer queues.

Usually flow control algorithms wait for congestion to occur before they are triggered. They are causal. However, due to the isolated nature of TCP connection - there are always only two parties involved - it is possible to predict congestions using the following algorithm:

- *Both parties exchange their receive socket and local buffer sizes*
- *Periodically, i.e. every N bytes, they send a control message marking the total number of bytes sent to the other party*
- *When the other party receives the control message it responds with a complementary control message indicating the received marker value. This acknowledges the reception of total bytes sent*
- *The difference between the total bytes sent and the last acknowledged marker received gives an indication of how full the other party's receive buffers are. This number should never exceed the total sum of receive buffer sizes.*

Flow control is needed only to optimize subscription messages back to the client (i.e. monitors). For other messages TCP flow control is sufficient.

A pvAccess implementation SHOULD implement flow control such that each endpoint should try to send as much monitoring data as it can subject to an upper limit calculated with respect to the amount of the other party's free receive buffer size. Were this limit to be reached, monitors would start piling up in the monitors' circular buffer queues.

Flow Control Example

The intention of flow control is to avoid having the following behavior, which typically results from pure TCP flow control:

- *Let's assume the client's Rx buffers are full.*
- *The server sends monitors until TCP detects the client's Rx buffer is full.*

- *After some time the client's Rx buffer is immediately emptied. This is a consequence of the fact that bulk reads are made from the socket rather than reading message by message (because OS calls are expensive).*
- *Server starts sending monitors until all the buffers are full (the server will fill all the buffers before the client actually processed received monitors!).*

Such situations as described above would result in monitors like the following (identified by their sequential number):

```
0 1 2 3 4 (buffers full) 7 8 9 10 11 12 (buffers full) 22 23 24 25 26
27 28 (buffers full)
```

Flow control can make this better:

```
0 1 2 3 4 (buffers full) 7 8 (buffers still full, but for less time
since the server would send only as much as the client can handle) 10
11 (...) 14 15 (...) 18 19
```

The result is more fluid and up-to-date arrival of monitors, which overcomes the combined problems of slow processing and large buffers.

Requiring flow control (in addition to already existing monitor queues) would add complexity to the protocol's implementation. It needs to be decided whether the above flow control should be specified as part of the normative specification, or only suggested non-normatively. At present, it is only suggested.

Channel Discovery

pvAccess uses a broadcast/multicast channel discovery mechanism using UDP; search messages are usually sent to broadcast addresses and servers hosting searched channels respond with a message containing their server address and port. In addition pvAccess transparently supports multicast, if an address is a multicast address the implementation SHOULD transparently handle it. That is, it should join the multicast group in order to receive multicast messages.

Possible future addition: UDP congestion control should be added to the specification to prevent the possibility of poor implementations flooding a network with UDP search messages. Currently a simple and robust algorithm is used in the reference implementation. The optimality of the algorithm should to be verified and added to this specification.

Communication Example

The following table illustrates messages sent between a client and a server where the client issues a get request on a channel.

Server		**Client**
	<----	searchRequest (UDP broadcast/multicast)
searchResponse (UDP unicast)	---->	
TCP/IP connection established		
setByteOrderControlMessage	---->	
connectionValidationRequest	---->	
	<----	connectionValidationResponse
	<----	createChannelRequest
createChannelResponse	---->	
	<----	channelGetRequestInit
channelGetResponseInit	---->	
	<----	channelGetRequest
channelGetResponse	---->	
	<----	. . .
. . .	---->	
	<----	destroyRequest
	<----	channelDestroyRequest
channelDestroyResponse	---->	

Communication Example.

Protocol Messages

The pvAccess protocol uses two protocol message types:

- Control messages. These include flow control and have no payload
- Application messages. These are the requests and their responses.

Each message consists of a message header and, optionally a message payload that immediately follows the header. Messages MUST BE aligned on a 64-bit boundary.

Every implementation of the protocol which purports to support this specification version of the protocol, MUST also support all prior specification versions of the protocol. Every implementation of the protocol MUST clearly indicate the most recent specification version to which it is conformant, using the version URLs above.

Message Header

Each protocol message has a fixed 8-byte header that MUST be encoded as if it were expressed by the following structure:

```
struct pvAccessHeader {
    byte magic;
    byte version;
    byte flags;
    byte messageCommand;
    int payloadSize;
};
```

The semantics of these message header components are given in the following table.

Member	Description
magic	pvAccess protocol magic code. This MUST always be 0xCA.
version	Protocol version.
flags	Message flags.
messageCommand	Message command (i.e. create, get, put, process, etc.).
payloadSize	Message payload size (non-aligned, in bytes).

pvAccess Header Members.

bit	Value	Description
0	0	Application message.
	1	Control message.

1,2,3	Unused, MUST be 0.	
5,4	00	Not segmented message.
	01	First messsage (of set of segmented messages).
	10	Last message (of set of segmented messages).
	11	Middle message (of set of segmented messages).
6	0	Message sent by client.
	1	Message sent by server.
7	0	Little endian byte order.
	1	Big endian byte order.

pvAccess Header Flags Description.

Between two segmented messages of the same set there MUST NOT be any other application message than the segmented message of the same set. Control messages are allowed to be in-between.

Alignment offset MUST be preserved between segmented messages, i.e. if the last sent byte of a segmented message is misaligned by 6 bytes to the 64-bit aligned start of the message reference point, then the next segmented message needs to insert 6 padding bytes at the start of the next segmented message payload.

Application Messages

This section describes the message payloads for application messages. Each subsection describes a single message command (*pvAccessHeader.messageCommand*).

"request" means a message sent by a client, and "response" means a message sent by a server.

In order to understand specific application messages it is helpful to be familiar with the EPICS V4 pvAccess Programmers Reference.

Most application messages below relate to the management of process variable channels. A process variable, or PV, is a dynamical quantity and its associated local processing semantics, as understood by process control systems. pvAccess has been designed to specifically integrate with process control systems, particularly EPICS V4, to provide an efficient interconnect for systems involved in the exchange of PV related information. Agent systems connect to a process control computer (via pvAccess) hosting PVs, by opening a "Channel" to each PV of interest. A channel is the temporal connection between pvAccess agents, with respect to one process variable.

All application message MUST be sent over the data transmission transport unless explicitly specified. TCP/IP is the transport in the reference implementation. A response message MUST be sent over the same transport as that on which the request was received.

Beacon (0x00)

Servers MUST broadcast or multicast beacons over UDP. Beacons are be used to announce new servers and server restarts.

```
struct beaconMessage {
    byte[12] guid;
    byte flags;
    byte beaconSequenceId;
    short changeCount;
    byte[16] serverAddress;
    short serverPort;
    string protocol;
    FieldDesc serverStatusIF;
    [if serverStatusIF != NULL_TYPE_CODE] PVField serverStatus;
};
```

Member	Description
guid	Server GUID (Globally Unique Identifier). MUST change every

	restart.
flags	reserved
beaconSequenceId	Beacon sequence ID (counter w/ rollover). Can be used to detect UDP routing problems.
changeCount	Count (w/ rollover) that changes every time server's list of channels changes.
serverAddressIPv6	Server address (e.g. for IP transports IPv6 or IPv6 encoded IPv4 address).
serverPort	Server port (e.g. for IP transport socket port where server is listening).
protocol	Protocol/transport name (e.g. "tcp" for standard pvAccess TCP/IP communication).
serverStatusIF	Optional server status Field description, NULL_TYPE_CODE MUST be used indicate absence of data.
serverStatus	Optional server data.

Beacon Message Members.

When a pvAccess server is started it MUST start emitting beacons. Clients MUST monitor all beacons. A beacon received from an as yet unknown serverAddress:serverPort MUST be interpreted as indicating that a new server has come online. A beacon with the same serverAdddress:serverPort address as one already received but has different globally unique ID (guid), MUST be interpreted as indicating that the server was restarted. In both cases a client SHOULD boost searching of not yet found channels. A client SHOUD also boost searching of not yet found channels when changeCount changes (this indicates that the server might host new channels). A client MAY disconnect old connections or wait until connection loss is detected (on failed Echo message send).

Each server transport instance SHOULD emit its own beacons. For example, if a server supports data transmission over TCP/IP and UDP/IP then these SHOULD both emit beacons. If the instances are tightly coupled, i.e. they have the same lifecycle and share the same channels, then only one server MAY emit beacons.

Due to the fact that UDP does not guarantee delivery, a server MUST send several beacons to notify that it is alive (e.g. 15 beacons with 1Hz period). After a longer period it MAY stop sending them, however it is recommended that it SHOULD continue merely with a low rate (e.g. one beacon per minute) to report serverStatus.

Beacons SHOULD not be used to report connection-valid status.

Connection validation (0x01)

A "connection validation" message MUST be the first application message sent from the server to a client when a TCP/IP connection is established. The message indicates that the server is ready to receive requests. The client MUST NOT send any messages on the connection until it has received a connection validation message from the server.

The purpose of the connection validation message is two-fold:

- It informs the client of the connection and protocol details.
- It prevents the client from writing a request message to its local transport buffers until after the server has acknowledged that it can actually process the request. This avoids a race condition caused by the server's TCP/IP stack accepting connections in its backlog while the server is in the process of shutting down. If the client were to send a request in this situation, the request would be lost but the client could not safely reissue the request because that might violate at-most-once semantics.

The connection validation message guarantees that a server is not in the middle of shutting down when the server's TCP/IP stack accepts an incoming connection, and so avoids the race condition.

The connection validation request and connection validation response messages are defined as follows:

```
struct connectionValidationRequest {
    int serverReceiveBufferSize;
    short serverIntrospectionRegistryMaxSize;
    string[] authNZ;                               // list of supported
authNZ;
};

// TODO new message code (from server) 0x09
struct connectionValidated {
    Status status;
};


// TODO new message code 0x05
struct authNZRequest {
    FieldDesc dataIF;
    [if dataIF != NULL_TYPE_CODE] PVField data;
};

struct authNZResponse {
    FieldDesc dataIF;
    [if dataIF != NULL_TYPE_CODE] PVField data;
};

// TODO new message code (from server) 0x06
```

```
struct aclChange {
    int clientChannelID;
    struct {
        int requestID;  // invalid ID is 0 (means for channel)
        BitSet[] rights;    // get has only one bit-set (readRights),
put-get has 2 (read and write), channel has one (allowed/dissallowed
request)
    } changes[];
};
```

```
struct connectionValidationResponse {
    int clientReceiveBufferSize;
    short clientIntrospectionRegistryMaxSize;
    short connectionQos;
    string authNZ;                          // selected authNZ plugin;
};
```

Member	Description
serverReceiveBufferSize	Server receive buffer size in bytes.
serverReceiveSocketBufferSize	Server socket buffer size in bytes.
serverIntrospectionRegistryMaxSize	Maximum number of introspection registry entries server is able to handle.

Connection Validation Request Message Members.

Member	Description
clientReceiveBufferSize	Client receive buffer size in bytes.
clientReceiveSocketBufferSize	Client socket buffer size in bytes.
clientIntrospectionRegistryMaxSize	Maximum number of introspection registry entries client is able to handle.
connectionQoS	Connection QoS parameters.

Connection Validation Response Message Members.

bit	Description
0-6	Priority level [0-100].
7	Unused, MUST be 0.
8	Low-latency priority.
9	Throughput priority.
10	Enable compression.
11-15	Unused, MUST be 0.

Connection QoS Parameters Description.

Each Quality of Service (QoS) parameter value REQUIRES a separate TCP/IP connection. If the Low-latency priority bit is set, this indicates clients should attempt to minimize latency if they have the capacity to do so. If the Throughput priority bit is set, this indicates a client similarly should attempt to maximize throughput. How this is achieved is implementation defined. The Compression bit enables compression for the connection . A matter for a future version of the specification should be whether a streaming mode algorithm should be specified.

Echo (0x02)

An Echo diagnostic message is usually sent to check if TCP/IP connection is still valid.

```
struct echoRequest {
    byte[] somePayload;
};

struct echoResponse {
    byte[] samePayloadAsInRequest;
};
```

Member	Description
somePayload	Arbitrary payload content, can be empty.

Echo request message members.

Member	Description
samePayloadAsInRequest	Same paylaod as in request message.

Echo response message members.

Search request (0x03)

A channel "search request" message SHOULD be sent over UDP/IP, however UDP congestion control SHOULD be implemented in this case. A server MUST accept this message also over TCP/IP.

```
struct searchRequest {
    int searchSequenceID;
    byte flags; // 0-bit for replyRequired, 7-th bit for "sent as
unicast" (1)/"sent as broadcast/multicast" (0)

    byte[3] reserved;
    // if not provided (or zero), the same transport is used for
responses
    // needs to be set when local broadcast (multicast on loop
interface) is done
    byte[16] responseAddress; // e.g. IPv6 address in case of IP based
transport, UDP
    short responsePort;        // e.g. socket port in case of IP based
transport

    string[] protocols;

    struct {
        int searchInstanceID;
        string channelName;
    } channels[];
};
```

Member	Description
`searchSequenceID`	Search sequence ID (counter w/ rollover), can be used by congestion control algorithms.
`replyRequired`	0x01 to force server to respond even if it does not host channel(s), 0x00 otherwise.
`protocol`	A set of allowed protocols to respond. Unrestricted if array is empty.
`searchInstanceID`	ID to be used to associate response with the following channel name.
`channelName`	Non-empty channel name, maximum length of 500 characters.

Search request message members.

The response to a search request is defined as messageCommand 0x04, see below.

Search response (0x04)

A "search response" message MUST be sent as the response to a search request (0x03) message.

```
struct searchResponse {
    byte[12] guid;
    int searchSequenceID;
    byte[16] serverAddress; // e.g. IPv6 address in case of IP based
transport
    short serverPort;        // e.g. socket port in case of IP based
transport
    string protocol;
    boolean found;
    int[] searchInstanceIDs;
};
```

Member	Description

searchSequenceID	Search sequence ID, same as specified in search request.
found	Flag indicating whether response contains IDs of found or not found channels.
serverAddressIPv6	Server address (e.g. in case of IP transport IP or IPv6 encoded IPv4 address).
serverPort	Server port (e.g. in case of IP trasnport socket port where server is listening).
protocol	Protocol name, "tcp" for standard pvAccess TCP/IP communication.
searchInstanceIDs	IDs, associated with names in the request, relevant to this response.

Search response message members.

A client MUST examine the protocol member field to verify it supports the given exchange protocol; if not, the search response is ignored.

Create channel (0x07)

A channel provides a communication path between a client and a server hosted "process variable."

Each channel instance MUST be bound only to one connection.

```
struct createChannelRequest {
    struct {
        int clientChannelID;
        string channelName;
    } channels[];
};

struct createChannelResponse {
    int clientChannelID;
    int serverChannelID;
    Status status;
    [if status.type == OK | WARNING] short accessRights;
};
```

Member	Description
clientChannelID	Client generated channel ID.
channelName	Name of the channel to be created, non-empty and maximum length of 500 characters.

Create channel request message members.

Member	Description
clientChannelID	Client generated channel ID, same as in request

serverChannelID	Server generated channel ID.
status	Completion status.
accessRights	Access rights (TBD).

Create channel response (per channel) message members.

NOTE: A server MUST store the clientChannelID and respond with its value in a destroyChannelMessage when a channel destroy request is requested, see below. A client uses the serverChannelID value for all subsequent requests on the channel. Agents SHOULD NOT make any assumptions about how given IDs are generated. IDs MUST be unique within a connection and MAY be recycled after a channel is disconnected.

Destroy channel (0x08)

A "destroy channel" message is sent to a server to destroy a channel that was previously created (with a create channel message).

```
struct destroyChannelRequest {
    int clientChannelID;
    int serverChannelID;
};

struct destroyChannelResponse {
    int clientChannelID;
    int serverChannelID;
};
```

Member	Description
clientChannelID	Client generated channel ID, same as in create request.
serverChannelID	Server generated channel ID, same as in create response.

Destroy channel request.

Member	Description
clientChannelID	Client generated channel ID, same as in create request.
serverChannelID	Server generated channel ID, same as in create response.
status	Completion status.

Destroy channel response.

If the request (clientChannelID, serverChannelID) pair does not match, the server MUST respond with an error status. The server MAY break its response into several messages.

NOTE: A server MUST send this message to a client to notify the client about server-side initiated channel destruction. Subsequently, a client MUST mark such channels as disconnected. If the client's interest in the process variable continues, it MUST start sending search request messages for the channel.

Channel get (0x0A)

A "channel get" set of messages are used to retrieve (get) data from the channel.

```
struct channelGetRequestInit {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x08 for INIT;
    FieldDesc pvRequestIF;
    PVField pvRequest;
};

struct channelGetResponseInit {
    int requestID;
    byte subcommand;
    Status status;
    [if status.type == OK | WARNING] FieldDesc pvStructureIF;
};
```

Member	Description
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Client generated request ID.
subcommand	0x08
pvRequestIF	pvRequest Field description.
pvRequest	pvRequest structure.

Channel get init request.

Member	Description
requestID	Request ID, same as in request message.
subcommand	0x08, same as in request message.
status	Completion status.
pvStructureIF	pvStructure (data container) Field description.

Channel get init response.

After a get request is successfully initialized, the client can issue actual get request(s).

```
struct channelGetRequest {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x40 for GET; additional 0x10 mask for DESTROY;
};
```

```
struct channelGetResponse {
    int requestID;
    byte subcommand;
    Status status;
    [if status.type == OK | WARNING] BitSet changedBitSet;
    [if status.type == OK | WARNING] PVField pvStructureData;
};
```

Member	Description
`serverChannelID`	Server generated channel ID, same as in create channel response.
`requestID`	Request ID, same as in init message.
`subcommand`	0x40 for GET, additional 0x10 mask for DESTROY.

Channel get request.

Member	Description
`requestID`	Request ID, same as in request message.
`subcommand`	Same as in request message.
`status`	Completion status.
`changedBitSet`	Changed BitSet for pvStructureData.
`pvStructureData`	Data structure.

Channel get response.

NOTE: if the DESTROY mask is applied, the server MUST destroy the request after the get response and the client MUST do the same after it receives the response.

Channel put (0x0B)

A "channel put" set of messages are used to set (put) data to the channel.

```
struct channelPutRequestInit {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x08;
    FieldDesc pvRequestIF;
    PVField pvRequest;
};

struct channelPutResponseInit {
    int requestID;
    byte subcommand;
    Status status;
    [if status.type == OK | WARNING] FieldDesc pvPutStructureIF;
};
```

Member	Description
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Client generated request ID.
subcommand	0x08
pvRequestIF	pvRequest Field description.
pvRequest	pvRequest structure.

Channel put init request.

Member	Description
requestID	Request ID, same as in request message.
subcommand	0x08, same as in request message.
status	Completion status.
pvPutStructureIF	pvPutStructure (data container) Field description.

Channel put init response.

After a put request is successfully initialized, the client can issue actual put request(s) on the channel.

```
struct channelPutRequest {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x00 for PUT; 0x10 mask for DESTROY;
    BitSet toPutBitSet;
    PVField pvPutStructureData;
};

struct channelPutResponse {
    int requestID;
    byte subcommand;
    Status status;
};
```

Member	Description
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Request ID, same as in init message.
subcommand	0x00 for PUT, additional 0x10 mask for DESTROY.
toPutBitSet	To-put BitSet for pvPutStructureData.
pvPutStructureData	Data to put structure.

Channel put request.

Member	Description
requestID	Request ID, same as in request message.
subcommand	Same as in request message.
status	Completion status.

Channel put response.

A "get-put" request retrieves the remote put structure. This MAY be used by user applications to show data that was set the last time by the application.

```
struct channelGetPutRequest {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x40;
};

struct channelGetPutResponse {
    int requestID;
    byte subcommand;
    Status status;
    [if status.type == OK | WARNING] PVField pvPutStructureData;
};
```

Member	Description
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Request ID, same as in init message.
subcommand	0x40.

Channel get put request.

Member	Description
requestID	Request ID, same as in request message.
subcommand	Same as in request message.
status	Completion status.
pvPutStructureData	Remote put data structure.

Channel get put response.

Channel put-get (0x0C)

A "channel put-get" set of messages are used to set (put) data to the channel and then immediately retrieve data from the channel. Channels are usually "processed" or "updated" by their host between put and get, so that the get reflects changes in the process variable's state.

```
struct channelPutGetRequestInit {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x08;
    FieldDesc pvRequestIF;
    PVField pvRequest;
};

struct channelPutGetResponseInit {
    int requestID;
    byte subcommand;
    Status status;
    [if status.type == OK | WARNING] FieldDesc pvPutStructureIF;
    [if status.type == OK | WARNING] FieldDesc pvGetStructureIF;
};
```

Member	Description
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Client generated request ID.
subcommand	0x08
pvRequestIF	pvRequest Field description.
pvRequest	pvRequest structure.

Channel put-get init request.

Member	Description
requestID	Request ID, same as in request message.
subcommand	0x08, same as in request message.
status	Completion status.
pvPutStructureIF	pvPutStructure (data container) Field description.
pvGetStructureIF	pvGetStructure (data container) Field description.

Channel put-get init response.

After a put-get request is successfully initialized, the client can issue actual put-get request(s) on the channel.

```
struct channelPutGetRequest {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x00 for PUT_GET; 0x10 mask for DESTROY;
    BitSet toPutBitSet;
    PVField pvPutStructureData;
};

struct channelPutGetResponse {
    int requestID;
    byte subcommand;
    Status status;
    [if status.type == OK | WARNING] PVField pvGetStructureData;
};
```

Member	Description
`serverChannelID`	Server generated channel ID, same as in create channel response.
`requestID`	Request ID, same as in init message.
`subcommand`	0x00 for PUT_GET, additional 0x01 mask for DESTROY.
`toPutBitSet`	To-put BitSet for pvPutStructureData.
`pvPutStructureData`	Data to put structure.

Channel put-get request.

Member	Description
`requestID`	Request ID, same as in request message.
`subcommand`	Same as in request message.
`status`	Completion status.
`pvGetStructureData`	Get data structure.

Channel put-get response.

A "get-put" request retrieves the remote put structure. This MAY be used by user applications to show data that was set the last time by the application.

```
struct channelGetPutRequest {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x80;
};

struct channelGetPutResponse {
    int requestID;
    byte subcommand;
    Status status;
    [if status.type == OK | WARNING] PVField pvPutStructureData;
};
```

Member	Description
`serverChannelID`	Server generated channel ID, same as in create channel response.
`requestID`	Request ID, same as in init message.
`subcommand`	0x80.

Channel get put request.

Member	Description
requestID	Request ID, same as in request message.
subcommand	Same as in request message.
status	Completion status.
pvPutStructureData	Remote put data structure.

Channel get put response.

A "get-get" request retrieves remote get structure. This MAY be used by user applications to show data that was retrieved the last time.

```
struct channelGetGetRequest {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x40;
};

struct channelGetGetResponse {
    int requestID;
    byte subcommand;
    Status status;
    [if status.type == OK | WARNING] PVField pvGetStructureData;
};
```

Member	Description
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Request ID, same as in init message.
subcommand	0x40.

Channel get get request.

Member	Description
requestID	Request ID, same as in request message.
subcommand	Same as in request message.
status	Completion status.
pvGetStructureData	Remote get data structure.

Channel get get response.

Channel monitor (0x0D)

The "channel monitor" set of messages are used by client agents to indicate that they wish to be asynchronously informed of changes in the state or values of the process variable of a channel. The subscribe mechanism is employed.

```
struct channelMonitorRequestInit {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x08;     // | 0x80 pipeline support  // TODO
    FieldDesc pvRequestIF;
    PVField pvRequest;
    if [subcommand & 0x80 == 0x80] int queueSize;
};

struct channelMonitorResponseInit {
    int requestID;
    byte subcommand;
    Status status;
    [if status.type == OK | WARNING] FieldDesc pvStructureIF;
};
```

Member	Description
`serverChannelID`	Server generated channel ID, same as in create channel response.
`requestID`	Client generated request ID.
`subcommand`	0x08
`pvRequestIF`	pvRequest Field description.
`pvRequest`	pvRequest structure.

Channel monitor init request.

Member	Description
`requestID`	Request ID, same as in request message.
`subcommand`	0x08, same as in request message.
`status`	Completion status.
`pvStructureIF`	pvStructure (data container) Field description.

Channel monitor init response.

The pvRequest structure SHOULD be used to specify monitor queue size and algorithm. How it may be used for those functions is not defined by pvAccess, and so would be implementation defined.

After monitor request is successfully initialized, the client can issue the actual monitor request(s).

The following messages MUST be used to start (resume) or stop (suspend) monitoring and to destroy monitor requests (subscription):

```
struct channelStartMonitorRequest {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x44;
};
```

```
struct channelStopMonitorRequest {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x04;
};

struct channelDestroyMonitorRequest {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x10;
};
```

Member	Description
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Client generated request ID.
subcommand	0x44 for START, 0x04 for STOP, 0x10 for DESTROY.

Channel monitor requests.

The response for monitor requests above has the following form:

```
struct channelMonitorResponse {
    int requestID;
    byte subcommand = 0x00;
    BitSet changedBitSet;
    PVField pvStructureData;
    BitSet overrunBitSet;
};
```

// TODO

```
struct channelMonitorReportQueueSize {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x80;
    int nfree;
};
```

Member	Description
requestID	Request ID, same as in monitor init request message.
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Client generated request ID.
subcommand	0x00.
changedBitSet	Changed BitSet for pvStructureData.
pvStructureData	Data structure.
overrunBitSet	BitSet indicating overrun fields.

Channel monitor response.

Channel array (0x0E)

A "channel array" set of messages are used to handle remote array values. Requests allow a client agent to: retrieve (get) and set (put) data from/to the array, and to change the array's length (number of valid elements in the array).

```
struct channelArrayRequestInit {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x08;
    FieldDesc pvRequestIF;
    PVField pvRequest;
};

struct channelArrayResponseInit {
    int requestID;
    byte subcommand;
    Status status;
    [if status.type == OK | WARNING] FieldDesc pvArrayIF;
};
```

Member	Description
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Client generated request ID.
subcommand	0x08
pvRequestIF	pvRequest Field description.
pvRequest	pvRequest structure.

Channel array init request.

Member	Description
requestID	Request ID, same as in request message.
subcommand	0x08, same as in request message.
status	Completion status.
pvArrayIF	pvArray (data container) Field description.

Channel array init response.

After an array request is successfully initialized, the client can issue the actual array request(s).

```
struct channelGetArrayRequest {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x40 mask for GET; 0x10 mask for DESTROY;
    size offset;
    size count;
};
```

```
struct channelGetArrayResponse {
    int requestID;
    byte subcommand;
    Status status;
    [if status.type == OK | WARNING] PVField pvArrayData;
};
```

Member	Description
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Request ID, same as in init message.
subcommand	0x40 for GET, additional 0x10 mask for DESTROY.
offset	Offset from the beginning of the array.
count	Number of elements requested, 0 means form offset to the end of the array.

Channel array get request.

Member	Description
requestID	Request ID, same as in request message.
subcommand	Same as in request message.
status	Completion status.
pvArrayData	Data array.

Channel array get response.

```
struct channelPutArrayRequest {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x00 mask for PUT; 0x10 mask for DESTROY;
    size offset;
    PVField pvArrayData;
};

struct channelPutArrayResponse {
    int requestID;
    byte subcommand;
    Status status;
};
```

Member	Description
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Request ID, same as in init message.
subcommand	0x00 for PUT, additional 0x10 mask for DESTROY.
offset	Offset from the beginning of the array.
pvArrayData	Subarray to be put.

Channel array put request.

Member	Description
requestID	Request ID, same as in request message.
subcommand	Same as in request message.
status	Completion status.

Channel array put response.

/// TODO GetLength is missing, fix the codes !!!

```
struct channelSetLengthRequest {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x80 mask for SET_LENGTH; 0x10 mask for DESTROY;
    size length;
};

struct channelSetLengthResponse {
    int requestID;
    byte subcommand;
    Status status;
};
```

Member	Description
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Request ID, same as in init message.
subcommand	0x40 for GET, additional 0x10 mask for DESTROY.
length	New length.

Channel array set length request.

Member	Description
requestID	Request ID, same as in request message.
subcommand	Same as in request message.
status	Completion status.

Channel array set length response.

Destroy request (0xF)

A "destroy request" messages is used destroy any request instance, i.e. an instance with requestID.

```
// destroys any request with given requestID
```

```
struct destroyRequest {
    int serverChannelID;
    int requestID;
};
```

Member	Description
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Request ID, same as in request init message.

Destroy request.

Channel process (0x10)

A "channel process" set of messages are used to indicate to the server that the computation actions associated with a channel should be executed. In the language of EPICS, this means that the channel should be "processed".

```
struct channelProcessRequestInit {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x08;
    FieldDesc pvRequestIF;
    [if serverStatusIF != NULL_TYPE_CODE] PVField pvRequest;
};

struct channelProcessResponseInit {
    int requestID;
    byte subcommand;
    Status status;
};
```

Member	Description
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Client generated request ID.
subcommand	0x08
pvRequestIF	Optional pvRequest Field description, NULL_TYPE_CODE is none.
pvRequest	Optional pvRequest structure.

Channel process init request.

Member	Description
requestID	Request ID, same as in request message.
subcommand	0x08, same as in request message.
status	Completion status.

Channel process init response.

After a process request is successfully initialized, the client can issue the actual process request(s).

```
struct channelProcessRequest {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x00 mask for PROCESS; 0x10 mask for DESTROY;
};

struct channelProcessResponse {
    int requestID;
    byte subcommand;
    Status status;
};
```

Member	Description
`serverChannelID`	Server generated channel ID, same as in create channel response.
`requestID`	Request ID, same as in init message.
`subcommand`	0x00 for PROCESS, additional 0x10 mask for DESTROY.

Channel proces request.

Member	Description
`requestID`	Request ID, same as in request message.
`subcommand`	Same as in request message.
`status`	Completion status.

Channel process response.

Get channel type introspection data (0x11)

A "get channel type introspection data" message is used to retrieve a channel's type introspection data, i.e. a description of all the channel's fields and their data types.

```
struct channelGetFieldRequest {
    int serverChannelID;
    int requestID;
    string subFieldName;  // entire record if empty
};

struct channelGetFieldResponse {
    int requestID;
    Status status;
    [if status.type == OK | WARNING] FieldDesc subFieldIF;
};
```

Member	Description
`serverChannelID`	Server generated channel ID, same as in create channel response.

requestID	Client generated request ID.
subFieldName	Name of the subfield to get or entire record if empty.

Get channel introspection data request.

Member	Description
requestID	Request ID, same as in request message.
status	Completion status.
subFieldIF	Requested field introspection data.

Get channel introspection data response.

Message (0x12)

A "message" message is used by a server to provide to a client human readable text regarding the status of a specific request. This message MUST NOT be used to report request completion status.

```
struct message {
    int requestID;
    byte messageType; // info = 0, warning = 1, error = 2, fatalError =
3
    string message;
};
```

Member	Description
requestID	Request ID.
messageType	Message type enum.
message	Message.

Message response.

Channel RPC (0x14)

The "channel RPC" set of messages are used to provide remote procedure call (RPC) support over pvAccess.

```
struct channelRPCRequestInit {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x08;
    FieldDesc pvRequestIF;
    PVField pvRequest;
};

struct channelRPCResponseInit {
    int requestID;
    byte subcommand;
    Status status;
};
```

Member	Description
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Client generated request ID.
subcommand	0x08
pvRequestIF	pvRequest Field description.
pvRequest	pvRequest structure.

Channel RPC init request.

Member	Description
requestID	Request ID, same as in request message.
subcommand	0x08, same as in request message.
status	Completion status.

Channel RPC init response.

After a RPC request is successfully initialized, the client can issue actual RPC request(s).

```
struct channelRPCRequest {
    int serverChannelID;
    int requestID;
    byte subcommand = 0x00 mask for RPC; 0x10 mask for DESTROY;
    FieldDesc pvStructureIF;
    PVField pvStructureData;
};

struct channelRPCResponse {
    int requestID;
    byte subcommand;
    Status status;
    [if status.type == OK | WARNING] FieldDesc pvResponseIF;
    [if status.type == OK | WARNING] PVField pvResponseData;
};
```

Member	Description
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Request ID, same as in init message.
subcommand	0x00 for RPC, additional 0x10 mask for DESTROY.
pvStructureIF	pvStructureData Field description.
pvStructureData	Argument data structure.

Channel RPC request.

Member	Description
requestID	Request ID, same as in request message.
subcommand	Same as in request message.
status	Completion status.
pvResponseIF	pvResponseDataField description.
pvResponseData	Response data structure.

Channel RPC response.

Cancel request (0x15)

A "cancel request" messages is used cancel any pending request, i.e. an instance with requestID.

```
// cancel any request with given requestID
struct cancelRequest {
    int serverChannelID;
    int requestID;
};
```

Member	Description
serverChannelID	Server generated channel ID, same as in create channel response.
requestID	Request ID, same as in request init message.

Cancel request.

Control Messages

This section describes the message payloads for control messages. Each subsection describes a single message command (*pvAccessHeader.messageCommand*).

Control messages have no payload and are used internally by the protocol, for instance to handle byte order management and flow control.

The payload size field contains control message specific values.

Mark Total Byte Sent (0x00)

The payload size field holds the value of the total bytes sent. The client SHOULD respond with an acknowledgment control message (0x01) as soon as possible.

Acknowledge Total Bytes Received (0x01)

The payload size field holds the acknowledge value of total bytes received. This must match the previously received marked value as described above.

Set byte order (0x02)

The 7-th bit of a header flags field indicates the server's selected byte order for the connection on which this message was received. Client MUST encode all the messages sent via this connection using this byte order.
The client's decoding byte order depends on the payload size field value as follows:

Payload Size Field Value	Meaning
`0x00000000`	Client MUST decode all the messages received via this connection using server's selected byte order.
`0xFFFFFFFF`	Client MUST decode all the messages sent received this connection as indicated by each message byte order flag.

Client Decoding

This MUST be the first message sent by a server when connection is established. For connection-less protocols this message is not sent and byte order is determined per message using its byte order flag.

NOTE: this message is byte order independent.

Echo request (0x03)

Diagnostic/test echo message. The receiver should respond with an Echo response (0x04) message with the same payload size field value.

Echo response (0x04)

Response to a echo request. The payload size field contains the same value as in the request message.

Future Protocol Changes/Updates

The following are known items that should be specified in future revisions:

- "one-phase" get/put/get-put/process
- immutable fields support, cache implemented for values (useful for enums)
- optimized packed Monitor responses
- bulk message transfer/trottle public API
- access rights
- etc.

Missing Aspects

The following aspects are missing in the current revision of the specification and will be specified in future revisions:

- structure/content of pvRequestIF/pvRequest fields
- offset and count fields of channelArray request should be of type 'size', however 'size' cannot be negative
- update Communication Example section to show messages

Bibliography

bib:caref

EPICS R3.14 Channel Access Reference Manual, J.O. Hill, R. Lange, 2002, http://www.aps.anl.gov/epics/base/R3-14/8-docs/CAref.html

bib:pvdatarefcpp

EPICS pvDataCPP [pvData C++ Programmers Reference Manual], M. Kraimer, 2011 under development, http://epics-pvdata.sourceforge.net/docbuild/pvDataCPP/tip/documentation/pvDataCPP.html

bib:pvdatarefjava

EPICS pvDataJava [pvData Java Programmers Reference Manual], M. Kraimer, 2011 under development, http://epics-pvdata.sourceforge.net/docbuild/pvDataJava/tip/documentation/pvDataJava.html

bib:ieee754wiki

IEEE 754-2008, Wikipedia article, April 2012, http://en.wikipedia.org/wiki/IEEE_754-1985