FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS

FIPA ACL Message Representation in Bit-Efficient Encoding Specification

Document title	FIPA ACL Message Representation in Bit-Efficient Encoding Specification		
Document number	XC00069C	**Document source**	FIPA Agent Management
Document status	Experimental	**Date of this status**	2000/07/25
Supersedes	FIPA00024		
Contact	fab@fipa.org		
Change history			
2000/07/25	Approved for Experimental		

Geneva, Switzerland

Foreword

The Foundation for Intelligent Physical Agents (FIPA) is an international organization that is dedicated to promoting the industry of intelligent agents by openly developing specifications supporting interoperability among agents and agent-based applications. This occurs through open collaboration among its member organizations, which are companies and universities that are active in the field of agents. FIPA makes the results of its activities available to all interested parties and intends to contribute its results to the appropriate formal standards bodies.

The members of FIPA are individually and collectively committed to open competition in the development of agent-based applications, services and equipment. Membership in FIPA is open to any corporation and individual firm, partnership, governmental body or international organization without restriction. In particular, members are not bound to implement or use specific agent-based standards, recommendations and FIPA specifications by virtue of their participation in FIPA.

The FIPA specifications are developed through direct involvement of the FIPA membership. The status of a specification can be either Preliminary, Experimental, Standard, Deprecated or Obsolete. More detail about the process of specification may be found in the FIPA Procedures for Technical Work. A complete overview of the FIPA specifications and their current status may be found in the FIPA List of Specifications. A list of terms and abbreviations used in the FIPA specifications may be found in the FIPA Glossary.

FIPA is a non-profit association registered in Geneva, Switzerland. As of January 2000, the 56 members of FIPA represented 17 countries worldwide. Further information about FIPA as an organization, membership information, FIPA specifications and upcoming meetings may be found at http://www.fipa.org/.

Contents

1 Scope

This document is part of the FIPA specifications and deals with message transportation between inter-operating agents. This document also forms part of the FIPA Agent Management Specification [FIPA00023] and contains specifications for:

- Syntactic representation of ACL in a bit-efficient form.

2 Bit-Efficient ACL Representation

This section defines the message transport syntax for a bit-efficient encoding which is expressed in standard EBNF format (see *Table 1*).

Note that this representation is not compatible with [FIPA00075].

Grammar rule component	Example
Terminal tokens are enclosed in double quotes	`"("`
Non-terminals are written as capitalised identifiers	`Expression`
Square brackets denote an optional construct	`[ "," OptionalArg ]`
Vertical bars denote an alternative between choices	`Integer | Float`
Asterisk denotes zero or more repetitions of the preceding expression	`Digit*`
Plus denotes one or more repetitions of the preceding expression	`Alpha+`
Parentheses are used to group expansions	`( A | B )*`
Productions are written with the non-terminal name on the left-hand side, expansion on the right-hand side and terminated by a full stop	`ANonTerminal = "terminal".`
`0x??` is a hexadecimal byte	`0x00`

Table 1: EBNF Rules

White space is not allowed between tokens.

2.1 Component Name

The name assigned to this component is:

```
fipa.acl.rep.bitefficient.std
```

2.2 Syntax

```
ACLCommunicativeAct     = Message.

Message                 = Header MessageType MessageParameter* EndofMsg.

Header                  = MessageId Version.

MessageId               = 0xFA
                        | 0xFB
                        | 0xFC.                       /* see comment 1 below */

Version                 = Byte.                       /* see comment 2 below */

EndofMsg                = EndOfCollection.

EndOfCollection         = 0x01.

MessageType             = PredefinedMsgType
                        | UserDefinedMsgType.         /* see comment 3 below */

UserDefinedMsgType      = 0x00 MsgTypeName.

MsgTypeName             = BinWord.

MessageParameter        = PredefinedParam
```

```
                        | UserDefinedMsgParam.        /* see comment 4 below */

UserDefinedMsgParam     = 0x00 ParameterName ParameterValue.

ParameterName           = BinWord.

ParamterValue           = BinExpression.

PredefinedMsgType       = 0x01                  /* accept-proposal */
                        | 0x02                  /* agree */
                        | 0x03                  /* cancel */
                        | 0x04                  /* cfp */
                        | 0x05                  /* confirm */
                        | 0x06                  /* disconfirm */
                        | 0x07                  /* failure */
                        | 0x08                  /* inform */
                        | 0x09                  /* inform-if */
                        | 0x0a                  /* inform-ref */
                        | 0x0b                  /* not-understood */
                        | 0x0c                  /* propagate */
                        | 0x0d                  /* propose */
                        | 0x0e                  /* proxy */
                        | 0x0f                  /* query-if */
                        | 0x10                  /* query-ref */
                        | 0x11                  /* refuse */
                        | 0x12                  /* reject-proposal */
                        | 0x13                  /* request */
                        | 0x14                  /* request-when */
                        | 0x15                  /* request-whenever */
                        | 0x16.                 /* subscribe */

PredefinedMsgParam      = 0x02 AgentIdentifier  /* :sender */
                        | 0x03 RecipientExpr    /* :receiver */
                        | 0x04 MsgContent       /* :content */
                        | 0x05 ReplyWithParam   /* :reply-with */
                        | 0x06 ReplyByParam     /* :reply-by */
                        | 0x07 InReplyToParam   /* :in-reply-to */
                        | 0x08 ReplyToParam     /* :reply-to */
                        | 0x09 Language         /* :language */
                        | 0x0a Encoding         /* :encoding */
                        | 0x0b Ontology         /* :ontology */
                        | 0x0c Protocol         /* :protocol */
                        | 0x0d ConversationID.  /* :conversation-id */

AgentIdentifier         = 0x02 AgentName
                          [Addresses]
                          [Resolvers]
                          (UserDefinedParameter)*
                          EndOfCollection.

AgentName               = BinWord.

Addresses               = 0x02 UrlCollection.

Resolvers               = 0x03 AgentIdentifierCollection.
```

```
UserDefinedParameter    = 0x04 BinWord BinExpression.

UrlCollection           = (Url)* EndofCollection.

Url                     = BinWord.

AgentIdentifierCollection
                        = (AgentIdentifier)* EndOfCollection.

RecipientExpr           = AgentIdentifierCollection.

MsgContent              = BinExpression.

ReplyWithParam          = BinExpression.

ReplyByParam            = BinDateTimeToken.

InReplyToParam          = BinExpression.

ReplyToParam            = RecipientExpr.

Language                = BinExpression.

Encoding                = BinExpression.

Ontology                = BinExpression.

Protocol                = BinWord.

ConversationID          = BinExpression.

BinWord                 = 0x10 Word 0x00
                        | 0x11 Index.

BinNumber               = 0x12 Digits           /* Decimal Number */
                        | 0x13 Digits.          /* Hexadecimal Number */

Digits                  = CodedNumber+.

BinString               = 0x14 String 0x00      /* New string literal */
                        | 0x15 Index            /* String literal from code table*/
                        | 0x16 Len8 ByteSeq     /* New ByteLengthEncoded string */
                        | 0x17 Len16 ByteSeq    /* New ByteLengthEncoded string */
                        | 0x18 Index            /* ByteLengthEncoded from code table*/
                        | 0x19 Len32 ByteSeq.   /* New ByteLengthEncoded string */

BinDateTimeToken        = 0x20 BinDate
                        | 0x21 BinDate TypeDesignator.

BinDate                 = Year Month Day Hour Minute Second Millisecond.
                              /* see comment 9 below */

BinExpression           = BinExpr
                        | 0xFF BinString.       /* See comment 10 below */

BinExpr                 = BinWord
```

```
                        | BinString
                        | BinNumber
                        | ExprStart BinExpr* ExprEnd.

ExprStart               = 0x60              /* Level down (i.e. '(' –character) */
                        | 0x70 Word 0x00    /* Level down, new word follows */
                        | 0x71 Index        /* Level down, word code follows */
                        | 0x72 Digits       /* Level down, number follows */
                        | 0x73 Digits       /* Level down, hex number follows */
                        | 0x74 String 0x00  /* Level down, new string follows */
                        | 0x75 Index^n      /* Level down, string code follows */
                        | 0x76 Len8 String  /* Level down, new byte string (1 byte) */
                        | 0x77 Len16 String /* Level down, new byte string (2 byte) */
                        | 0x78 Len32 String /* Level down, new byte string (4 byte) */
                        | 0x79 Index^n.     /* Level down, byte string code follows */

ExprEnd                 = 0x40              /* Level up (i.e. ')' –character) */
                        | 0x50 Word 0x00    /* Level up, new word follows */
                        | 0x51 Index        /* Level up, word code follows */
                        | 0x52 Digits       /* Level up, number follows */
                        | 0x53 Digits       /* Level up, hexadecimal number follows */
                        | 0x54 String 0x00  /* Level up, new string follows */
                        | 0x55 Index        /* Level up, string code follows */
                        | 0x56 Len8 String  /* Level up, new byte string (1 byte) */
                        | 0x57 Len16 String /* Level up, new byte string (2 byte) */
                        | 0x58 Len32 String /* Level up, new byte string (4 byte) */
                        | 0x59 Index.       /* Level up, byte string code follows */

ByteSeq                 = Byte*.

Index                   = Byte
                        | Short.               /* See comment 7 below */

Len8                    = Byte.                /* See comment 8 below */

Len16                   = Short.               /* See comment 8 below */

Len32                   = Long.                /* See comment 8 below */

Year                    = Byte Byte.

Month                   = Byte.

Day                     = Byte.

Minute                  = Byte.

Second                  = Byte.

Millisecond             = Byte Byte.

Word                    = /* as in [FIPA00070]  */

String                  = /* as in [FIPA00070] */

CodedNumber             = /* See comment 5 below */
```

```
TypeDesignator          = /* as in [FIPA00070] */
```

2.3 Using Dynamic Code Tables

The transport syntax can be used with or without dynamic code table. Using dynamic code tables is an optional feature, which gives more compact output but might not be appropriate if communicating peers does not have sufficient memory (for example, in case of low-end PDAs or smart phones).

To use dynamic code tables the encoder inserts new entries (for example, `Word`, `String`, etc.) into a code table while constructing bit-efficient representation for ACL message. The code table is initially empty and whenever a new entry is added to the code table, the smallest available code number is allocated to it. There is no need to transfer these index codes explicitly over the communication channel. Once the code table becomes full and a new code needs to be added, the sender first removes `size>>3`[1] entries from the code table using a Least Recently Used (LRU) algorithm and then adds a new entry to code table. For example, should the code table size be 512 entries, 64 entries are removed. Correspondingly the decoder removes entries from the code table when it receives a new entry from the encoder.

The size of the code table, if used, is between 256 (2^8) and 65536 (2^{16}) entries. The output of this code table is always one or two bytes (one byte only when the code table size is 2^8). Using two-byte output code wastes some bits, but allows for much faster parsing of messages. The code table is unidirectional, that is, if sender A adds something to the code table when sending a message to B, then B cannot use this code table entry when sending a message back to A.

2.4 Notes on the Grammar Rules

1. The first byte defines the message identifier. The identifier byte can be used to separate bit-efficient ACL messages from (for example) string-based messages and separate different coding schemes. The value `0xFA` defines a bit-efficient coding scheme without dynamic code tables and the value `0xFB` defines a bit-efficient coding scheme with dynamic code tables. The message identifier `0xFC` is used when dynamic code tables are being used, but the sender does not want to update code tables (even if message contains strings that should be added to code table).

2. The second byte defines the version number. The version number byte contains the major version number in the upper four bits and minor version number in the lower four bits. This specification defines version 1.0 (coded as `0x10`).

3. All message types defined in this specification have a predefined code. If an encoder sends an ACL message with a message type which has no predefined code, it must use the extension mechanism which adds a new message type into code table (if code tables are being used).

4. All message parameters defined in this specification have a predefined code. If a message contains a user defined message parameter, an extension mechanism is used (byte `0x00`) and new entry is added to code table (if code table is used).

5. Numbers are coded by reserving four bits for each digit in the number's ASCII representation, that is, two ASCII numbers are coded into one byte. *Table 1* shows a 4-bit code for each number and special codes that may appear in ASCII coded numbers.

   If the ASCII presentation of a number contains odd number characters, the last four bits of the coded number are set to zero (the `Padding` token), otherwise an additional `0x00` byte is added to end of coded number. If the number to be coded is integer, decimal number, or octal number, the identifier byte `0x12` is used. For hexadecimal numbers, the identifier byte `0x13` is used. Hexadecimal numbers are converted to integers before coding (the coding scheme does not allow characters from a through `f` to appear in number form).

   Numbers are never added to a dynamic code table.

[1] Right shifted by 3 bit positions – approximately 10%.

Token	Code		Token	Code
Padding	0000		7	1000
0	0001		8	1001
1	0010		9	1010
2	0011		+	1100
3	0100		E	1101
4	0101		-	1110
5	0110		.	1111
6	0111			

6. Table 1: Binary Representation of Number Tokens

7. `Index` is a pointer to code table entry and its size (in bits) depends on the code table size. If the code table size is 256 entries, the size of the index is one byte; otherwise its size is two bytes (represented in network byte order).

8. `Byte` is a one-byte code word, `Short` is a short integer (two bytes, network byte order) and `Long` is a long integer (four bytes, network byte order).

9. Dates are coded as numbers, that is, four bits are reserved for each ASCII number (see comment 5 above). Information whether the type designator is present or not, is coded into identifier byte. These fields always have static length (two bytes for year and milliseconds, one byte for other components).

10. None of the actual content of the message (the information contained in the `:content` parameter of the ACL message) is coded nor are any of its components are added to a code table.

3 References

[FIPA00023] FIPA Agent Management Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00023/

[FIPA00067] FIPA Agent Message Transport Service Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00067/

[FIPA00070] FIPA ACL Message Representation in String Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00070/

[FIPA00075] FIPA Agent Message Transport Protocol for IIOP Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00075/