Request for Comments: 713 Jack Haverty (JFH@MIT-DMS)
NIC #34739 Apr 1976

I. ABSTRACT

A mechanism is defined for use by message servers in transferring data between hosts. The mechanism, called the MSDTP, is defined in terms of a model of the process as a translation between two sets of items, the abstract entities such as ’strings’ and ’integers’, and the formats used to represent such data as a byte stream.

A proposed organization of a general data transfer mechanism is described, and the manner in which the MSDTP would be used in that environment is presented.

II. REFERENCES

Black, Edward H., "The DMS Message Composer", MIT Project MAC, Programming Technology Division Document SYS.16.02.

Burchfiel, Jerry D., Leavitt, Elsie M., Shapiro, Sonya and Strollo, Theodore R., compilers, "Tenex Users' Guide", Bolt Beranek and Newman, Cambridge, Mass., May 1971, revised January 1975, Descriptive sections on the TENEX subsystems: MAlLER, p. 116-11; MAlLSTAT, p. 118-119; READMAIL, p. 137; and SNDMSG, p. 165-170.

Haverty, Jack, "Communications System Overview", MIT Project MAC, Programming Technology Division Document SYS.16.00.

Haverty, Jack, "Communications System Daemon Manual", MIT Project MAC, Programming Technology Division Document SYS.16.01.

ISI Information Automation Project, "Military Message Processing System Design," Internal Project Documentation (Out of Print), Jan. 1975

Message Services Committee, "Interim Report", Jan. 28, 1975

Mooers, Charlotte D., "Mailsys Message System: Manual For Users", Bolt Beranek and Newman, Cambridge, Mass., June 1975 (draft).

Myer, Theodore H., "Notes On The BBN Mail System", Bolt Beranek and Newman, November 8, 1974.

Myer, Theodore H., and Henderson, D. Austin, "Message Transmission Protocol", Network Working Group RFC 680, NIC 32116, April 30, 1975.

Postel, Jon, "The PCPB8 Format", NSW Proposal, June 5, 1975

Tugender, R., and D. R. Oestreicher, "Basic Functional Capabilities for a Military Message Processing Service," ISI?RR-74-23., May 1975

Vezza, Al, "Message Services Committee Minority Report", Jan. 1975

III. OVERVIEW

This document describes a mechanism developed for use by message servers communicating over an eight-bit byte-oriented network connection to move data structures and associated data-typing information. It is presented here in the hope that it may be of use to other projects which need to transfer data structures between dissimilar hosts.

A set of abstract entities called PRIMITIVE ITEMS is enumerated. These are intended to include traditional data types of general utility, such as integers, strings, and arrays.

A mechanism is defined for augmenting the set of abstract data entities handled, to allow the introduction of application-specific data, whose format and semantics are understood by the application programs involved, but which can be transmitted using common coding facilities. An example might be a data structure called a 'file specification', or a 'date'. Abstract data entities defined using this mechanism will be termed SEMANTIC ITEMS, since they are typically used to carry data having semantic content in the application involved.

Semantic and primitive items are collectively referred to simply as ITEMS.

The protocol next involves the definition of the format of the byte stream used to convey items from machine to machine. These encodings are described in terms of OBJECTS, which are the physical byte streams transmitted.

To complete the protocol, the rules for translating between objects and items are presented as each object is defined.

An item is transmitted by being translated into an object which is transmitted over the connection as a stream of bytes to the receiver, and reconstructed there as an item. The protocol mechanism may thus be viewed as a simple translator. It enumerates a set of abstract entities, the items, which are known to programmers, a set of entities in byte-stream format, the objects, and the translation rules for conversion between the sets. A site implementing the MSDTP would typically provide a facility to convert between objects and the local representation of the various items handled. Applications using the MSDTP define their interactions using items, without regard to the actual formats in which such items are represented at various machines. This permits programs to handle higher-level concepts such as a character string, without concern for its numerous representational formats. Such detail is handled by the MSDTP.

Finally, a discussion of a general data transfer mechanism for communication between programs is presented, and the manner in which the particular byte-oriented protocol defined herein would be used in that environment is discussed.

Terminology, as introduced, is defined and highlighted by capitalizing.

IV. PRIMITIVE DATA ITEMS

The primitive data items include a variety of traditional, well-understood types, such as integers and strings. Primitive data items will be presented using mnemonic names preceded by the character pair "p-", to serve as a reminder that the named object is primitive.

These items may be represented in various computer systems in whatever fashion their programmers desire.

IV.1 -- Set Of Primitive Items

The set of primitive items defined includes p-INT, p-STRING, p-STRUC, p-BITS, p-CHAR, p-BOOL, p-EMPTY, and p-XTRA.

Since the protocol was developed primarily for use in message services, items such as p-FLOAT are not included since they were unnecessary. Additional items may be easily added as necessary.

A p-INT performs the traditional role of representing integer numbers. A p-BITS (BIT Stream) item represents a bit stream. The two possible p-BOOL (BOOLean) items are used to represent the logical values of *TRUE* and *FALSE*. The single p-EMPTY item is used to, for example, indicate that a given field of a message is empty. It is provided to act as a place-holder, representing ’no data’, and appears as *EMPTY*.

The p-STRUC (STRUCture) item is used to group together a collection of items as a single value, maintaining the ordering of the elements, such as a p-STRUC of p-INTs.

A p-CHAR is a single character. The most common occurrence of character data, however, will be as p-STRINGs. A p-STRING should be considered to be a synonym for a p-STRUC containing only p-CHARs. This concept is important for generality and consistency, especially when considering definitions of permissible operations on structures, such as extracting subsequences of elements, etc.

Four p-XTRA items, which can be transmitted in a single byte, are made available for higher level protocols to use when a frequently used datum is handled which can be represented just by its name. An example would be an acknowledgment between two servers. Using p-XTRAs to represent such data permits them to be handled in a single byte. There are four possible p-XTRA items, termed *XTRA0*, *XTRA1*, *XTRA2*, and *XTRA3*. These may be assigned meanings by user protocols as desired.

IV.2 -- Printing Conventions

The following printing conventions are introduced to facilitate discussion of the primitive items.

When a specific instance of a primitive data item is presented, it will be shown in a traditional representation for that kind of data. For example, p-INTs are shown as sequences of digits, e.g. 100, p-STRINGs, as sequences of characters enclosed in double-quote characters, for example "ABCDEF".

As shown above, the two possible p-BOOL items are shown as *TRUE* or *FALSE*. The object p-EMPTY appears as *EMPTY*. A bit stream, i.e. p-BITS, appears as a stream of 1s and 0s enclosed in asterisks, for example *100101001*. A p-CHAR will be presented as the character enclosed in single quote characters, e.g., 'A'.

P-STRUCs are printed as the representations of their elements, enclosed in parentheses, for example (1 2 3 4) or ("XYZ" "ABC" 1 2) or ((1 2 3) "A" "B"). Note that because p-STRINGs are simply a class of p-STRUCs assigned a special name and printing format for brevity and convenience, the items "ABC" and ('A' 'B' 'C') are identical, and the latter format should not be used.

To present a generic p-STRUC, as in specifying formats of the contents of something, the items are presented as a mnemonic name, optionally followed by a colon and the permissible types of values for that datum. When one of several items may appear as the value for some component, the permissible ones appear separated by vertical-bar characters. For example, p-INT|p-STRING represents a single item, which may be either a p-INT or a p-STRING.

To represent a succession of items, the Kleene star convention is used. The specification p-INT[*] represents any number of p-INTs. Similarly, p-INT[3,5] represents from 3 to 5 p-INTs, while p-INT[*,5] specifies up to 5 and p-iNT[5,*] specifies at least 5 p-INTs.

For example, a p-STRUC which is used to carry names and numbers might be specified as follows.

(name:p-STRING number:p-INT)

In discussing items in general, when a specific data value is not intended, the name and types representation may be used, e.g., offset:p-INT to discuss an 'offset' which has a numeric value.

V. SEMANTIC ITEM MECHANISM

The semantic item mechanism provides a means for program designers to use a variety of application-specific data items.

This mechanism is implemented using a special tagged structure to carry the data type information as well as the actual components of the particular semantic item. For discussion purposes. Such a special p-STRUC will be termed a p-EDT (Extended Data Type).

When p-EDTs are transferred, their identity as a p-EDT is maintained. So that an applications program receives the corresponding semantic item instead of a simple p-STRUC. A p-EDT is identical to a p-STRUC in all other respects.

V.1 -- Format of p-EDTs

A prototypical p-EDT follows. It is printed as if it were a normal p-STRUC. Since p-EDTs are converted to semantic items for presentation to the user, a p-EDT will never be used except in this protocol definition.

(type:p-INT|p-STRING version:p-INT com1:any com2:any ...)

The first element, the 'type' is generally a p-INT, and is used to identify the particular type of semantic item. Types are assigned numeric codes in a controlled fashion. The type may alternatively be specified by a p-STRING, to permit development of new data types for possible later assignment of codes. Each type has an equivalent p-STRING name. These may be used interchangeably as 'type' elements, primarily to maintain upward compatibility.

The second element of a p-EDT is always an p-INT, the 'version', and specifies the exact format of the particular datum. A semantic item may undergo several revisions of its internal structure. Which would be evident through assigning different versions to each revision.

Successive components. The 'com' elements, if any. carry the actual data of the semantic item. As each semantic item is defined, conventions on permissible values and interpretation of these components are presented. Such definitions may use any types of items to specify the format of the semantic item. Use of lower level concepts, such as objects, in these definitions is prohibited.

Semantic items will be printed as the mnemonic for the type involved, preceded by the character pair "s-", to signify that the data item is handled by this mechanism.

V.2 -- Printing Conventions

A semantic item is represented as if it were a p-STRUC containing only the components, if any, but preceded by the semantic type name and a # character. The version number is assumed to be 1 if unspecified. For later versions, the version number is attached to the type name, as in, for example, FILE-2 to represent version 2 of the FILE data type.

For example, a semantic item called a 'file specification' might be defined, containing two components, a host number and pathname. A specific instance of such an item might appear as #FILE(69 "DIRECTORY.NAME-OF-FILE"), while a generic s-FILE might be presented as the following.

#FILE(host:p-INT|p-STRING pathname:p-STRING)

the item, which may be either a p-INT or p-STRING, and 'pathname' is the second component, which must be a p-STRING. The full definition would present interpretation rules for these components.

VI. ENCODING OBJECTS

This section presents the set of objects which are used to represent items as byte streams for inter-server transmission. Objects will be presented using mnemonic type-names preceded by the character pair "b-", indicating their existence only as byte streams.

All servers are required to be capable of decoding the entire set of objects. Servers are not required to transmit certain objects which are available to improve channel efficiency.

The encodings are designed to facilitate programming and efficiency of the receiving decoder. In all cases, the type and length in bytes of objects is supplied as the first information sent. This characteristic is important for ease of implementation. The type information permits a decoder to be constructed in a modular fashion. The most important advantage of including size information is that the receiver always knows how many bytes it must read to discover what to do next, and knows when each object terminates. This requirement avoids many potential problems with timing and synchronization of processes.

Two varieties of objects are defined. The first will be called ATOMIC, and includes objects used to efficiently encode the most common data. The second variety is termed NON-ATOMIC, and is used to encode larger or less common items.

In all cases, a data object begins with a single byte, which will be termed the TYPE-BYTE, a field of which contains the type code of the object. The following bytes, if any, are interpreted according to the type involved.

VI.1 -- Presentation Conventations

In discussing formats of bytes, the following conventions will be employed. The individual bits of a byte will be referenced by using capital letters from A to H, where A signifies the highest order bit, and H the lowest. The entire eight bit value, for example, could be referred to as ABCDEFGH. Similarly, subfields of the byte will be identified by such sequences. The CDEF field specifies the middle four bits of a byte.

In referring to values of fields, binary format will be used, and small letters near the end of the alphabet will be used to identify particular bits for discussion. For example, we might say that the BCD field of a byte contains a specifier for some type, and define its value to be BCD=11z. In discussions of the specifier usage, we could refer to the cases where z=l and where z=0, as shorthand notation to identify BCD=111 and BCD=110, respectively.

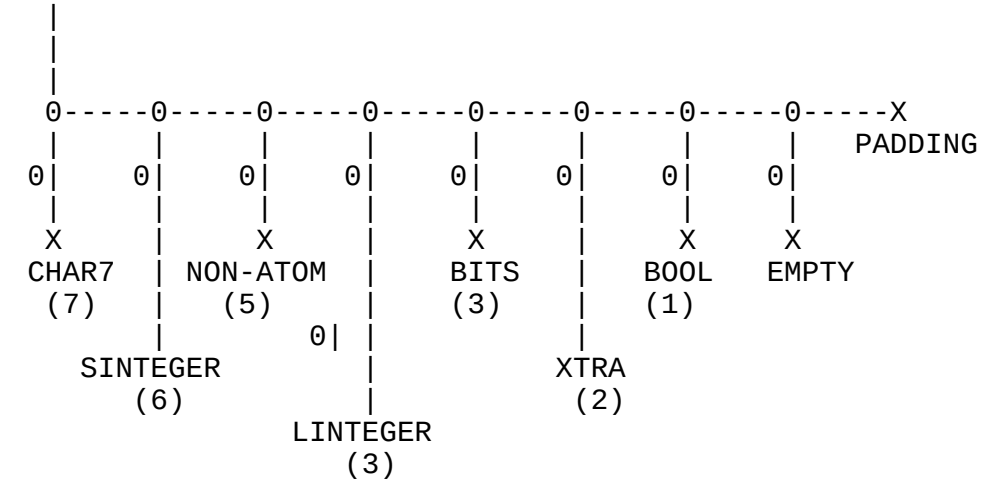
V1.2 -- Type-Byte Bit Assignment

To assist in understanding the assignment of the various type-byte values, the table and graph below are included, showing representations of the eight bits.

```
OXXXXXXX -- CHAR7 (CHARacter, 7 bit)
10XXXXXX -- SINTEGER (Small INTEGER)
l10XXXXX -- NON-ATOM (NON-ATOMic objects)
11100XXX -- LINTEGER (Large INTEGER)
11101XXX -- reserved
11110XXX -- SBITSTR (Short BIT STReam)
111110XX -- XTRA (eXTRA single-byte objects)
1111110X -- BOOL (BOOLean)
11111110 -- EMPTY (EMPTY data item)
11111111 -- PADDING (unused byte)
```

In each case, the bits identified by X's are used to contain information specific to the type involved. These are explained when each type is defined.

An equivalent tree representation follows, for those who prefer it.



Type-Byte Bit Assignment Scheme

This picture is interpreted by entering at the top, and taking the appropriate branch at each node to correspond to the next bit of the type-byte, as it is scanned from left to right. When a type is assigned, the branch terminates with an "X' and the name of the type of the object, with the number of remaining bits in parentheses. The individual object definitions specify how these bits are used for that particular type.

V1.3 -- Atomic Objects

Atomic objects are identified by specific patterns in a type-byte. Receiving servers must be capable of recognizing

and handling all atomic types, since the size of the object is not explicitly present in a uniform fashion.

```
================================
| Atomic Object: B-CHAR7       |
================================
```

The b-CHAR7 (CHARacter 7 bit) object is introduced to handle transmission of characters, in 7-bit ASCII format. Since the vast majority of message-related data involves such objects, they are designed to be very efficient in transmission. Other formats, such as eight bit values, can be introduced as non-atomic objects. The format of a b-CHAR7 follows:

A=0 identifying the b-CHAR7 data type
BCDEFGH=tuvwxyz containing the character code

The tuvwxyz objects contain the ASCII code of the character. For example, transmission of a "space' (ASCII code 32, 40 octal) would be accomplished by the following byte.

00100000
ABCDEFGH

A=0 to identify this byte as a b-CHAR7. The remaining bits contain the 7 bit code, octal 40, for space.

A b-CHAR7 standing alone is presented as a p-CHAR. Such occurrences will probably be rare if they are used at all. The most common use of b-CHAR7's is as elements of b-USTRUCs used to transmit p-STRINGS, as explained later.

```
============================
| Atomic Object: B-SINTEGER |
============================
```

The b-SINTEGER (Small INTEGER) object is used to transmit very small positive integers, of values up to 64. It always translates to an p-INT, and any p-INT between 0 and 63 may be encoded as a b-SINTEGER for transmission. The format of an b-SINTEGER follows.

AB=10 identifying the object as a b-SINTEGER
CDEFGH=uvwxyz containing the actual number

For example, to transmit the integer 10 (12 octal), the following byte would be transmitted:

10001010
ABCDEFGH

AB=10 to specify a b-SINTEGER. The remaining six bits contain the number 10 expressed in binary.

```
============================
| Atomic Object: B-SINTEGER |
============================
```

The b-SINTEGER (Large INTEGER) object is used to transmit p-INTs to any precision up to 64 bits. It is always translated as a p-INT. Sending servers are permitted to choose either b-SINTEGER or b-SINTEGER format for transmission of numbers, as appropriate. When possible, b-SINTEGERs can be used for better channel efficiency. The format of a b-SINTEGER follows:

ABCDE=11100 specifying that this is a b-SINTEGER.
FGH=xyz containing a count of number of bytes to follow.

The xyz bits are interpreted as a number of bytes to follow which contain the actual binary code of the the integer in 2's complement format. Since a zero-byte integer is disallowed, the pattern xyz=000 is interpreted as 1000, specifying that 8 bytes follow. The number is transmitted with high-order bits first. This format permits transmission of integers as large as 64 bits in magnitude.

For example, if the number 4096 (10000 octal) is to be transmitted, the following sequence of bytes would be sent:

11100010 00010000 00000000
ABCDEFGH ---actual data---

ABCDE=11100, identifying this as a b-LINTEGER, E=0, specifying a positive number, and FGH=010, specifying that 2 bytes follow, containing the actual binary number.

```
============================
| Atomic Object: B-SBITSTR |
============================
```

The b-SBITSTR (Short BIT STReam) object is used to transmit a p-BITS of length 63 or less. For longer bit streams, the non-atomic object b-LBITSTR may be used. The format of a b-SBITSTR follows.

ABCDE=11110 specifying the type as b-SBITSTR
FGH=xyz specifying the number of bytes following.

The xyz value specifies the number of additional bytes to be read to obtain the bit stream values. As in the case of b-SINTEGER, the value xyz=000 is interpreted as 1000, specifying that 8 bytes follow.

To avoid requiring specification of exactly the number of bits contained, the following convention is used. The first data byte is scanned from left to right until the first 1 bit is encountered. The bit stream is defined to begin with the immediately following bit, and run through the last bit of the last byte read. In other words, the bit stream is 'right-adjusted' in the collected bytes, with its left end delimited by the first "on' bit.

For example, to send the bit stream *001010011* (9 bits), the following bytes are transmitted.

```
11110010 00000010 01010011
ABCDEhij klmnopqr stuvwxyz
```

The hij=010 value specifies that two bytes follow. The q bit, which is the first 1 bit encountered, identifies the start of the bit stream as being the r bit. The rstuvwxyz bits are the bit stream being handled.

```
=========================
| Atomic Object: b-BOOL |
=========================
```

The b-BOOL (BOOLean) object is used to transmit p-BOOLs. The format of b-BOOL objects follows.

```
ABCDEFG=1111110 specifying the type as
b-BOOL
H=z specifying the value
```

The two possible translations of a b-BOOL are *FALSE* and *TRUE*.

```
11111100 represents *FALSE*
11111101 represents *TRUE*
ABCDEFGz
```

if z=0, the value is FALSE, otherwise TRUE.

```
=======================================
| Atomic Object: B-EMPTY |
=======================================
```

The b-EMPTY object type is used to transmit a 'null' object, i.e. an *EMPTY*. The format of an b-EMPTY follows.

```
ABCDEFGH=11111110 specifying *EMPTY*
```

```
=========================
| Atomic Object: B-XTRA |
=========================
```

The b-XTRA objects are used to carry the four possible p-XTRA items, i.e., *XTRA0*, *XTRA1*, *XTRA2*, and *XTRA3*. These four items correspond to the binary coding of the remaining two bits after the b-XTRA type code bits. The format of a b-XTRA follows.

ABCDEF=111110 to specify the type b-XTRA
GH=yz to identify the particular p-XTRA item carried

The GH bits of the byte are decoded to produce a particular p-XTRA item, as follows.

GH=00 -- *XTRA0*
GH=01 -- *XTRA1*
GH=10 -- *XTRA2*
GH=11 -- *XTRA3*

The b-XTRA object is included to provide the use of several single-byte data items to higher levels. These items may be assigned by individual applications to improve the efficiency of transmission of several very frequent data items. For example, the message services protocols will use these items to convey positive and negative acknowledgments, two very common items in every interaction.

```
=======================================
| Atomic Object: B-PADDING
=======================================
```

This object is anomalous, since it represents really no data at all. Whenever it is encountered in a byte stream in a position where a type-byte is expected, it is completely ignored, and the succeeding byte examined instead. Its purpose is to serve as a filler in byte streams, providing servers with an aid in handling internal problems related to their specific word lengths, etc. The encoders may freely use this object to serve as padding when necessary.

All b-PADDING data objects exist only within an encoded byte stream. They never cause any data item whatsoever to be presented externally to the coder module. The format of a b-PADDING follows.

ABCDEFGH=11111111

Note that this does not imply that all such ’null’ bytes in a stream are to be ignored, since they could be encountered as a byte within some other type, such as b-LINTEGER. Only bytes of this format which, by their position in the stream, appear as a ’type’ byte are to be ignored.

VI.4 -- Non-Atomic Objects

Non-atomic objects are are always transmitted preceded by both a single type byte and some small number of size byte(s). The type byte identifies that the data object concerned is of a non-atomic type, as well as uniquely specifying the particular type involved. All non-atomic objects have type byte values of the following form.

ABC=110 specifying that the object is non-atomic
DEFGH=vwxyz specifying the particular type of object

The vwxyz value is used to specify one of 31 possible non-atomic types. The value vwxyz=00000 is reserved for use in future expansion.

In all non-atomic data objects, the byte(s) following the type-byte specify the number of bytes to follow which contain the data object. In all cases, if the number of bytes specified are processed, the next byte to be seen should be another type-byte, the beginning of the next object in the stream.

The number of bytes containing the object size information is variable. These bytes will be termed the SIZE-BYTES. The first byte encountered has the following format.

A=s specifying the manner in which the size information is encoded
BCDEFGH=tuvwxyz specifying the size, or number of bytes containing the size

The tuvwxyz values supply a positive binary number. If the s value is a one, the tuvwxyz value specifies the number of bytes to follow which should be read and concatenated as a binary number, which will then specify the size of the object. These bytes will appear with high order bits first. Thus, if s=1, up to 128 bytes may follow, containing the count of the succeeding data bytes, which should certainly be sufficient.

Since many non-atomic objects will be fairly short, the s=0 condition is used to indicate that the 7 bits contained in tuvwxyz specify the actual data byte count. This permits

objects of sizes up to 128 bytes to be specified using one size-information byte. The case tuvwxyz=0000000 is interpreted as specifying 128 bytes.

For example, a data object of some non-atomic type which requires 100 (144 octal) bytes to be transmitted would be sent as follows.

```
110XXXXX -- identifying a specific
non-atomic object
01100100 -- specifying that 100 bytes follow
.
.
data -- the 100 data bytes
.
.
```

Note that the size count does not include the size-specifier byte(s) themselves, but does include all succeeding bytes in the stream used to encode the object.

A data object requiring 20000 (47040 octal) bytes would appear in the stream as follows.

```
110XXXXX -- identifying a specific
non-atomic object
10000010 -- specifying that the next 2 bytes
contain the stream length
01001110 -- first byte of number 20000
00100000 -- second byte
.
.
data -- 20,000 bytes
.
.
```

Interpretation of the contents of the 20000 bytes in the stream can be performed by a module which knows the specific format of the non-atomic type specified by DEFGH in the type-byte.

The remainder of this section defines an initial set of non-atomic types, the format of their encoding, and the semantics of their interpretation.

Non-atomic Object: B-LBITSTR

The b-LBITSTR (Long BIT Stream) data type is introduced to transmit p-BITS which cannot be handled by a b-SBITSTR. A b-LBITSTR may be used to transmit short p-BITS as well. Its format follows.

11000001 size-bytes data-bytes
ABCDEFGH

ABC=110 identifies this as a non-atomic object.
DEFGH=00001 specifies that it is a b-LBITSTR. The standard sizing information specifies the number of succeeding bytes. Within the data-bytes, the first object encountered must decode to a p-INT. This number conveys the length of the bit stream to follow. The actual bit stream begins with the next byte, and is left-adjusted in the byte stream. For example to encode *101010101010*, the following b-LBITSTR could be used, although a b-SBITSTR would be more compact.

11000001 -- identifies a b-LBITSTR
00000010 -- b-SINTEGER, to specify length
10001100 -- size = 2
10101010 -- first 8 data bits
10100000 -- last 4 data bits

==========
| Non-atomic Object: B-STRUC |
==========

The b-STRUC (STRUCture) data type is used to transmit any p-STRUC. The translation rules for converting a b-STRUC into a primitive item are presented following the discussion of b-REPEATs. The b-STRUC format appears as follows.

11000010 size-bytes data-bytes
ABCDEFGH

ABC=110 identifies this as a non-atomic type.
DEFGH=00010 specifies that the object is a b-STRUC. Within the data-bytes stream, objects simply follow in order. This implies that the b-STRUC encoder and decoder modules can simply make use of recursive calls to a standard encoder/decoder for processing each element of the b-STRUC.

Note that any type of object is permitted as an element of a b-STRUC, but the size information of the b-STRUC must include all bytes used to represent the elements.

Containment of b-STRUCs within other b-STRUCs is permitted to any reasonable level. That is, a b-STRUC may contain as an element another b-STRUC, which contains another b-STRUC, and so on. All servers are requires to handle such containment to at least a minimum depth of three.

Examples of encoded structures appear in a later section.

Non-atomic Object: B-EDT

A b-EDT is the object used as the carrier for p-EDTs in transmission of semantic items. It is functionally identical to a b-STRUC, but has a different type code to permit it to be identified and converted to a semantic item instead of a p-STRUC. The format of a b-EDT follows.

```
11000011 size-bytes data-bytes
ABCDEFGH
```

As with all non-atomic types, ABC=110 to identify this as such, and DEFGH=00011 to specify a b-EDT. The objects in the data-bytes are decoded as for b-STRUCs. However, the first object must decode to a p-iNT or p-STRING and the second to a p-INT, to conform to the format of p-EDTs.

Non-atomic Object: b-REPEAT

The b-REPEAT object is never translated directly into an item. It is legal only as an component of an enclosing b-STRUC, b-USTRUC, b-EDT, or b-REPEAT. A b-REPEAT is used to concisely specify a set of elements to be treated as if they appeared in the enclosing structure in place of the b-REPEAT. This provides a mechanism for encoding a sequence of identical data items or patterns efficiently for transmission.

A common example of this would be in transmission of text, where line images containing long sequences of spaces, or pages containing multiple carriage-return, line-feed pairs, are often encountered. Such sequences could be encoded as an appropriate b-REPEAT to compact the data for transmission. The format of a b-REPEAT is as follows.

```
11000100   -- identifyIng the object as a
                b-REPEAT
size-bytes -- the standard non-atomic object
                size information
countspec  -- an object which translates to a p-INT
.
.
data -- the objects which define the pattern
.
.
```

The ’countspec’ object must translate to an p-INT to specify the number of times that the following data pattern should be repeated in the object enclosing the b-REPEAT.

The remaining objects in the b-REPEAT constitute the data pattern which is to be repeated. The decoding of the enclosing structure will be continued as if the data pattern objects appeared 'countspec' times in place of the b-REPEAT. Zero repeat counts are permitted, for generality. They cause no objects to be simulated in the enclosing structure.

An encoder does not have to use b-REPEATs at all, if simplicity of coding outweighs the benefits of data compression. In message services, for example, an encoder might limIt itself to only compressing long text strings. It is important for compatibility, however, to have the ability in the decoders to handle b-REPEATs.

```
==============================
| Non-atomic Object: B-USTRUC |
==============================
```

The b-USTRUC (Uniform Structure) object type is provided to enable servers to convey the fact that a p-STRUC being transferred contains items of only a single type. The most common example would involve a b-USTRUC which translates to a p-STRUC of only p-CHARs, and hence may be considered to be a p-STRING. Servers may use this information to assist them in decoding objects efficiently. No server is required to generate b-USTRUCs.

The internal construction of a b-USTRUC is identical to that of a b-STRUC, except for the type-byte. The format of a b-USTRUC follows.

```
11000101 size-bytes data-bytes
ABCDEFGH
```

ABC=110 to identify a non-atomic object. DEFGH=00101 specifies the object as a b-USTRUC.

```
==============================
| Non-atomic Object: B-STRING |
==============================
```

The b-STRING object is included to permit explicit specification of a structure as a p-STRING. This information will permit receiving servers to process the incoming structure more efficiently. A b-STRING is formatted similarity to a b-USTRUC, except that its type-byte identifies the object as a b-STRI/NG. The normal sizing information is followed by a stream of bytes which are interpreted as b-CHAR7s, Ignoring the high-order bit. The format of a b-STRING follows.

```
11000110 size-bytes data-bytes
ABCDEFGH
```

ABC=110 to identify a non-atomic object. DEFGH=00110 specifies the object as a b-STRING.

VI.5 -- Structure Translation Rules

A b-STRUC is translated into a p-STRUC. This is performed by translating each object of the b-STRUC Into its corresponding item, and saving it for inclusion In the p-STRUC being generated. A b-USTRUC is handled similarly, but the coding programs may utilize the information that the resultant p-STRUC will contain items of uniform type. The preferred method of coding p-STRINGS is to use b-USTRUCs.

If all of the elements of the resultant p-STRUC are p-CHARs, it is presented to the user of the decoder as a p-STRING. A p-STRING should be considered to be a synonym for a p-STRUC containing only characters. It need not necessarily exist at particular sites which would present p-STRUCs of p-CHARs to their application programs

The object b-REPEAT is handled in a special fashion when encountered as an element. When this occurs, the data pattern of the b-REPEAT is translated into a sequence of items, and that sequence is repeated in the next higher level as many times as specified in the b-REPEAT. Therefore, b-REPEATS are legal only as elements of a surrounding b-STRUC, b-USTRUC, b-EDT, or b-REPEAT.

In encoding a p-STRUC or p-STRING for transmission, a translator may use b-REPEATs as desired to effect data compression, but their use is not mandatory. Similarly, b-STRINGS may be used, but are not mandatory.

A b-EDT is translated into a p-EDT to identify it as a carrier for a semantic item. Otherwise, it is treated identically to a b-STRUC.

VI.6 -- Translation Summary

The following table summarizes the possible translations between primitive items and objects.

```
p-INT    <--> b-LINTEGER, b-SINTEGER
p-STRING <--> b-STRING, b-STRUC, b-USTRUC
p-STRUC  <--> b-STRING, b-STRUC, b-USTRUC
p-BITS   <--> b=SBITSTR, b-LBITSTR
p-CHAR   <--> b-CHAR7
p-BOOL   <--> b-BOOL
p-EMPTY  <--> b=EMPTY
p-XTRA   <--> b-XTRA
p-EDT    <--> b-EDT (all semantic items)
-none-   <--> b-PADDING
-none-   <--> b-REPEAT (only within structure)
```

Note that all semantic items are represented as p-EDTs which always exist as b-EDTs in byte-stream format.

The following stream transmits a b-STRUC containing 3 b-SINTEGERs, with values 1, 2, and 3, representing a p-STRUC containing three p-INTs, i.e. (1 2 3).

```
11000010 -- b-STRUC
00000011 -- size=3
10000001 -- b-SINTEGER=1
10000010 -- b-SINTEGER=2
10000011 -- b-SINTEGER=3
```

The next example represents a b-STRUC containing the characters X and Y, followed by the b-LINTEGER 10, representing a p-STRUC of 2 p-CHARs and a p-INT, i.e., ('X' 'Y' 10). Note that the p-INT prevents considering this a p-STRING.

```
11000010 -- b-STRUC
00000100 -- size=4
01011000 -- b-CHAR7 'X'
01011001 -- b-CHAR7 'Y'
11100001 -- b-LINTEGER
00001010 -- 10
```

Note that a better way to send this p-STRUC would be to represent the integer as a b-SINTEGER, as shown below.

```
11000010 -- b-STRUC
00000011 -- size=3
01011000 -- b-CHAR7 'X'
01011001 -- b-CHAR7 'Y'
10001010 -- b-SINTEGER=10
```

The next example shows a b-STRUC of b-CHAR7s. It is the translation of the b-STRING "HELLO".

```
11000010 -- b-STRUC
00000101 -- size=5
01001000 -- b-CHAR7 'H'
01000101 -- b-CHAR7 'E'
01001100 -- b-CHAR7 'L'
01001100 -- b-CHAR7 'L'
01001111 -- b-CHAR7 'O'
```

This datum could also be transmitted as a b-STRING. Note that the character bytes are not necessarily b-CHAR7s, since the high-order bit is ignored.

```
11000110 -- b-STRING
00000101 -- size=5
01001000 -- 'H'
01000101 -- 'E'
01001100 -- 'L'
01001100 -- 'L'
01001111 -- 'O'
```

To encode a p-STRING containing 20 carriage-return line-feed pairs, the following b-STRUC containing a b-REPEAT could be used.

```
11000010 -- b-STRUC
00000101 -- size=5
11000100 -- b-REPEAT
00000011 -- size=3
10010100 -- count, b-SINTEGER=20
00001101 -- b-CHAR7, "CR'
00001010 -- b-CHAR7, 'IF'
```

To encode a p-STRUC of p-INTs, where the sequence contains a sequence of thirty 0's preceded by a single 1, the following b-STRUC could be used.

```
11000010 -- b-STRUC
00000110 -- size=6
10000001 -- b-SINTEGER=1
11000100 -- b-REPEAT
00000010 -- size=2
10011110 -- count, b-SINTEGER=30
10000000 -- b-SINTEGER=0
```

VII. A GENERAL DATA TRANSFER SCHEME

This section considers a possible scheme for extending the concept of a data translator into an multi-purpose data transfer mechanism.

The proposed environment would provide a set of primitive items, including those enumerated herein but extended as necessary to accommodate a variety of applications. Communication between processes would be defined solely in terms of these items, and would specifically avoid any consideration of the actual formats in which the data is transferred.

A repertoire of translators would be provided, one of which is the MSDTP machinery, for use in converting items to any of a number of transmission formats. Borrowing a concept from radio terminology, each translator would be analogous to a different type of modulation scheme, to be used to transfer data through some communications medium. Such media could be an eight-bit byte-oriented connection, 36-bit connection, etc. and conceivably have other distinguishing features, such as bandwidth, cost, and delay. For each media which a site supports, it would provide its programmers with a module for performing the translations required.

Certain media or translators might not handle various items. For example, the MSDTP does not handle items which might be termed p-FLOATs, p-COMPLEXs, p-ARRAY, and so on. In addition, the efficiency of various media for transfer of specific items may differ drastically. MSDTP, for example, transfers data frequently used in message handling very efficiently, but is relatively poor at transfer of very large or deep tree structures.

Available at each site as a process or subroutine package wouLd be a module responsible for interfacing with its counterpart at the other end of the media. These modules would use a protocol, not yet defined, to match their capabilities, and choose a particular media and translator, when more than one exists, for transfer of data items.

Such a facility could totally insulate applications from need to consider encoding formats, machine differences, and so on, as well as eliminate duplication of effort in producing such facilities for every new project which requires them. In addition, as new translators or media are introduced, they would become immediately available to existing users without reprogramming.

Implementation of such a protocol should not be very difficult or time-consuming, since it need not be very sophisticated in choosing the most appropriate transfer mechanism in initial implementations. The system is inherently upward-compatible and easily expandable.