Picture Interchange Coding (PIC)

—

Functional Specification and Encoding of Profile '2D'

F. Kappe *†

February 1989

Abstract

This documentation contains detailed specification of the **Picture Interchange Coding (PIC)** file format. PIC is a flexible format suitable for the encoding of graphical and related data.

The PIC data format has been designed for and is used in conjunction with EDEN-based interactive graphics editors. The functional specification is closely related to that of CGM, CGI and PHIGS.

The document presents an overview of the basic concepts of EDEN and PIC, the functional specification of PIC, and the coding specification of profile '2D'.

*IIG, Institutes for Information Processing Graz, Graz University of Technology and Austrian Computer Society (OCG)

†reformatted on Oct 6, 1992

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 The EDEN Editor Environment

Interactive graphics editors are used for various applications today (think of Computer Aided Design (CAD), Computer Aided Instruction(CAI), Videotex (Vtx) or Desk Top Publishing (DTP)). From both the user's and the software developer's point of view it is desirable to have a common framework for graphics editing covering many applications.

EDEN (short for **ED**itor **EN**vironment) is a generic concept for object-oriented interactive graphics editing ([6], [14]): Many different applications can be created using EDEN as their common nucleus (see figure 1).

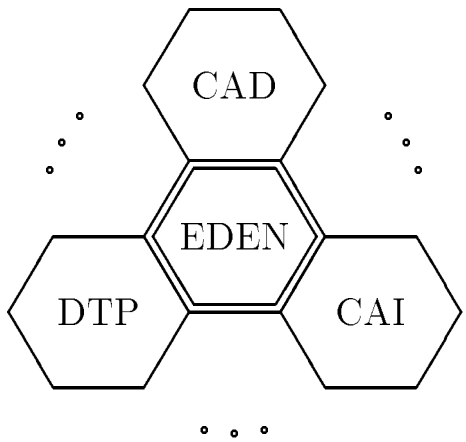

Figure 1: *EDEN as a common nucleus for various applications*

1.2 The Need for PIC

Obviously pictures and segments generated by EDEN-based applications have to be stored somehow. Usually there exists an application-specific file format for this purpose, but use of that format may have disadvantages (for instance, the hierarchical segment structure might be lost, the encode/decode process is slow etc.). Besides, an application-independent file format is desirable to interchange pictures across applications. Therefore EDEN supports both an application-independent file format called *Picture Interchange Coding (PIC)* and the encoder/decoder concept providing an interface to application-dependent file formats (see figure 2):

Throughout the remaining document, the routine that writes PIC files will be called the *PIC Writer* and the routine that reads PIC files will be called the *PIC Reader*.

1.3 Relation to CGM and PHIGS CSS

The Computer Graphics Metafile (CGM) standardized by ISO ([11]) provides a file format suitable for the storage and retrieval of picture information in three different encodings. Unfortunately, CGM does not support the powerful segment concept of EDEN, and the extension of CGM for private purposes using the *Escape* and *Application Data* elements is somewhat clumsy, so the use of CGM for the application-independent interchange of EDEN picture data is not feasible.
However, PIC is closely related to CGM in the following sense:

- The graphic primitives and associated attributes of CGM are also found in profile '2D' of PIC.
- The actual encoding of PIC primitives is similar to the CGM binary encoding ([11, part 3]).

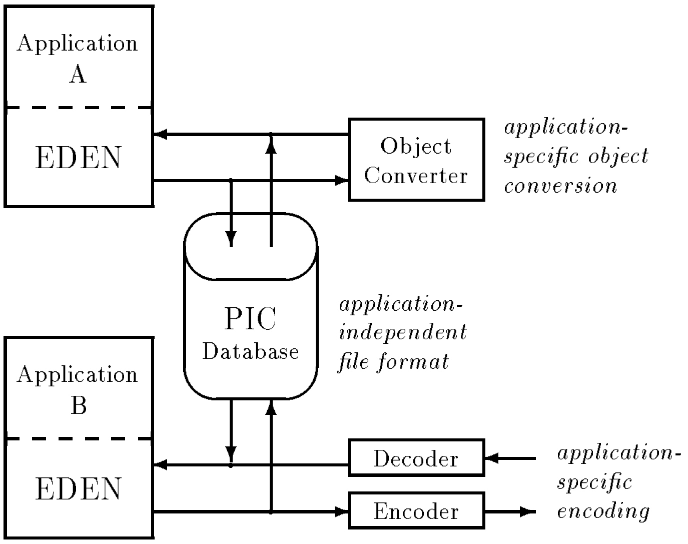

Figure 2: *EDEN interfacing the World*

Therefore, the conversion of CGM to PIC and vice versa can easily be done by the use of Encoder/Decoder pairs (see figure 2), although there are substantial coding differences:

- PIC supports more powerful concepts concerning segments (see section 2.2) and attributes (see section 2.3).
- CGM allows a variety of selectable precisions, specification modes and representation modes that were avoided in PIC for the following reasons:
  - A large number of coding options would make the PIC Reader more complex and therefore slower, which is undesirable.
  - It introduces context-sensitivity, which is also undesirable. For instance, how could a segment using atoms with *Colour Specification Mode = Indexed* be incorporated into a segment that uses *Colour Specification Mode = Direct* instead?

The EDEN concept of composing hierarchically structured pictures is also found in the PHIGS standard proposed by ISO ([13]). Segments (called *structures* in PHIGS notation) are organized as acyclic directed graphs, called *segment networks* (see figure 3). A PIC file may contain one ore more segment networks. However, the primitives of PHIGS are not part of profile '2D' of PIC.

PHIGS stores structure networks in the *Centralized Structure Store (CSS)*, that corresponds to the *PIC Filing System* discussed in section 10.

1.4 Profiles

The main emphasis of PIC is the extensibility of the format. In particular, PIC adapts to the needs of different EDEN-based, special-purpose graphics editors. As easily as EDEN allows the extension of the set of supported objects, PIC allows the storage and retrieval of them.

Usually different EDEN-based graphics editors need different sets of objects, attributes and other capabilities. These sets are called *PIC profiles*. Profile '2D' is defined in this document and is a 'foundation profile'. It contains the concepts, objects and attributes of the EDEN nucleus, that are automatically incorporated into every EDEN-based graphics editor. The '2D' profile supports static, two-dimensional pictures consisting of graphic primitives found in CGM together with CGM Addendum 1 ([12]) and CGI ([10]).
Other profiles, very likely to be defined in the future, are:

- A profile suitable for storage of videotex images of different standards ([2], [3], [4], [14]).

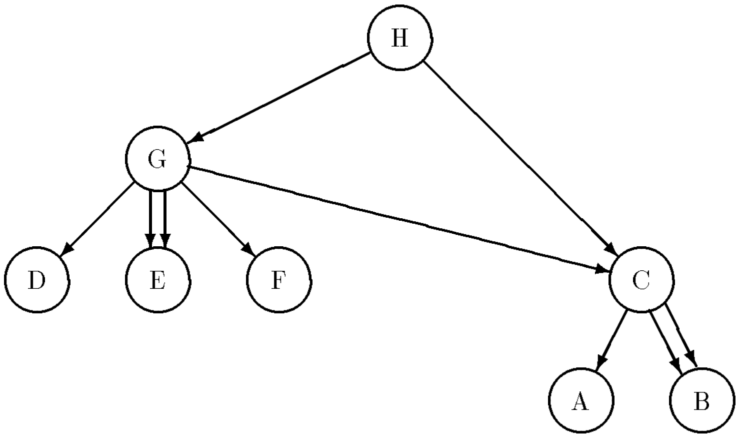

Figure 3: *An EDEN segment network*

- A profile for storage of Computer Aided Instruction (CAI) courseware ([7]).
- A '3D' profile.
- A profile suitable for handling of raster data.
- A profile for 2D- and 3D-animation.

Object converters (see figure 2) are used to convert PIC Files to different profiles (if possible, without loss of information).

2 Functional Specification

2.1 Atoms and Compound Objects

EDEN provides three different sets of objects, namely Atoms, Compound Objects and Segments.

Atoms are the basic objects that all other objects consist of. The built-in atoms are the graphic primitives of CGI, also covered by CGM plus CGM Addendum 1. Profile '2D' of PIC supports storage and retrieval of the atoms listed in table 1. A detailed specification of these atoms may be found in ([10, part 3]).

Group	**Atoms**	**Attributes**
Line	Polyline Disjoint Polyline Circular Arc 3 Point Circular Arc Centre Circular Arc Centre Backwards Elliptical Arc	Line Type Line Width Line Colour
Marker	Polymarker	Marker Type Marker Size Marker Colour
Fill Area	Polygon Polygon Set Rectangle Circle Circular Arc 3 Point Close Circular Arc Centre Close Ellipse Elliptical Arc Close	Interior Style Fill Colour Hatch Index Pattern Index Fill Reference Point Pattern Size Edge Type Edge Width Edge Colour Edge Visibility
Text	Simple Text Simple Restricted Text	Text Font Index Text Precision Character Expansion Factor Character Spacing Text Colour Character Height Character Orientation Text Path Text Alignment Character Set Index Alternate Character Set Index
Image	Cell Array Pixel Array	

Table 1: *Profile '2D' atoms and related attributes*

Compound objects can be displayed using atoms and segments, but need more functionality than segments. Profile '2D' supports three compound objects: The *Closed Figure, Compound Text* and *Compound Restricted Text* as defined in ([10, part 3], [12]).

PIC stores Atoms and Compound Objects with their attributes bound to them, in a way similar to the representation in the computer (see sections 5,6).

2.2 Segments

Segments are re-usable sub-images usually created interactively by the EDEN user. EDEN and PIC support two types of segments:

- **Global Segments** are stored like pictures. They may be referenced by any other picture. Modification of a global segment affects all pictures (or segments) referencing that segment. The PIC Filing System is responsible for the integrity of the segment network (e.g. ensures that no global segment that is still referenced by other segments may be deleted).

- **Local Segments** are accessible only from the picture (segment) they are defined in. Local Segments are stored within that picture and are not accessible from outside. As segments may be nested, a block structure similar to the nesting of functions in block-structured programming languages (like Algol68, Pascal) is created.

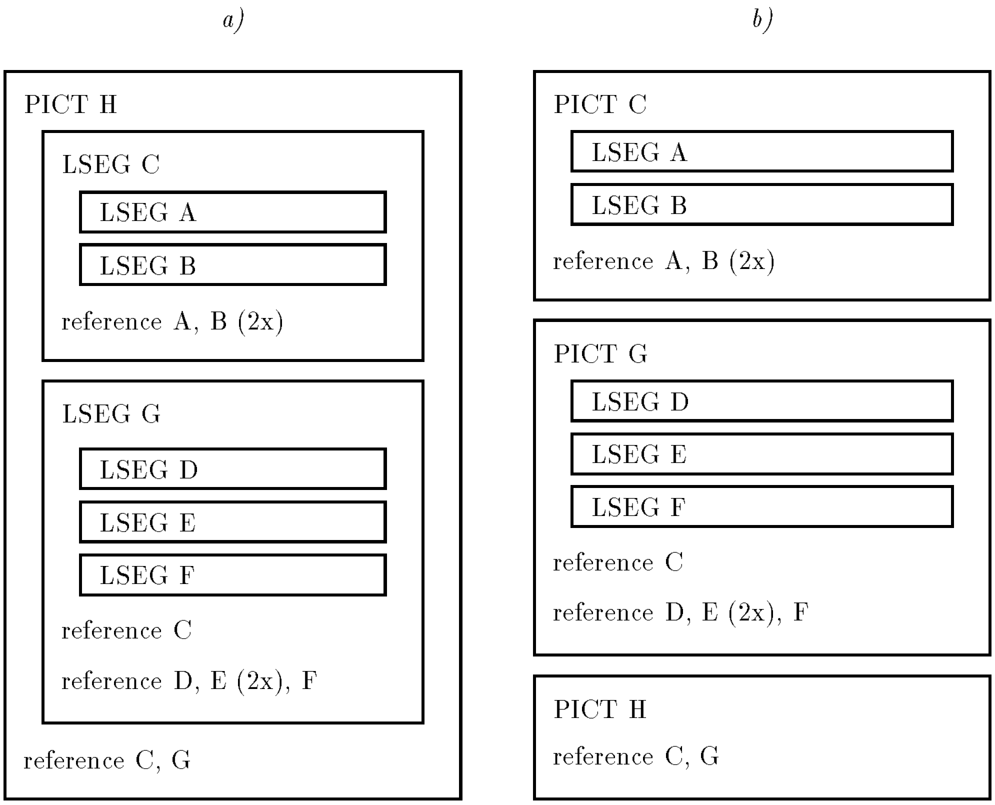

Figure 4: *Storing the segment network of Figure 3 in different ways*

Figure 4 shows two ways of storing the segment network depicted in figure 3. In Figure 4a) only local segments are used. The picture H contains segments C and G, which in turn contain some other segments. Note two things:

- Segment C is referenced by segment G, but also by picture H. Therefore segment C must not be stored local to segment G.

- The PIC Writer ensures that all local segments are defined (stored) before they are referenced. In our example, segment C has to be stored prior to segment G, so G can reference C. Because the segment network is assumed to be an acyclic graph, such an order of segments always exists.

In figure 4b) segments C and G are stored as global segments (pictures), so they may be referenced by other pictures. C, G and H may be stored in one file or in different files, the order within the PIC file is arbitrary. However, defining global instead of local segments introduces overhead to the PIC Filing System.

2.3 Attributes

The attributes of EDEN objects may be bound directly to the object. In this case the attributes are said to be *individual*. PIC stores individual attributes together with the geometrical aspects of an object (in contrast to CGM, where *Set-Primitives* are used to set attributes affecting following display primitives).

EDEN (and therefore PIC) also supports *bundled* attributes. The *Aspect Source Flag (ASF)* determines whether an attribute is bundled or individual. When the ASF for an attribute is bundled, PIC stores the name of a *variable*, that holds the actual attribute value.

To realize this, PIC contains functions to set variables to a specific value of any type or to refer to another variable. The variable names are 32-bit entities, that could be viewed as memory addresses, bundle table indices or 4-character names.
This *variable concept* allows the following features:

a. The *Bundle Table* concept of CGM/CGI, GKS ([9]) and PHIGS can be simulated, with a practically indefinite (2^{32}) number of bundle table entries.

b. The *Attribute Inheritance* concept of PHIGS and CGI can be simulated. A segment may, for instance, use a line colour that was set in the referencing segment.

c. Segments may be parametrized. For instance, a segment might be referenced by something like

```
Segment A (COL1=15, COL2=BLUE);
```

where `COL1` and `COL2` are variables used by segment A, and `BLUE` is another variable used by the referencing segment.

2.4 Coordinates

Coordinates are stored in *Virtual Device Coordinates (VDC)*, in a manner similar to CGM. The *VDC Extent* specifies the range of coordinates in a picture (or segment). Specification of values outside the VDC extent is permitted. It is intended that the visible portion of an image is contained within the VDC extent. For example, an EDEN application may display the bounding rectangle of a segment using the VDC extent of the segment definition. An application may choose to describe pictures in coordinates that map directly to the Device Coordinates (DC) of a target device, but still are displayed correctly on other devices. Whether the mapping of VDC to DC is isotropic or not is implementation-dependent.
In order to keep PIC Readers simple (and fast), there are differences when compared to CGM:

- In PIC, the direction of the positive x and positive y axes is fixed. The minimum x and y values of the VDC extent define the lower left corner and the maximum values the upper right corner of the drawing.
- All coordinates are stored as single-precision (32-bit), real values (see section 4.9).

3 Overall Encoding Structure

A file format for encoding pictures can be designed with a number of different objectives:

a. *Compactness*: The encoding provides a highly compact file, suitable for systems with restricted storage capacity or transfer bandwidth (Example: CGM character encoding ([11, part 2]), CEPT videotex standards ([3],[4])).

b. *Processing Speed*: The encoding uses binary data formats similar to the representation used within the computer in order to minimize processing overhead in reading/writing the data file (Example: CGM binary encoding ([11, part 3]), PIC).

c. *Human Readability*: The representation of the data is easy to read, type and edit using a standard text editor (Example: CGM Clear text encoding ([11, part 4])).

d. *Extensibility*: The encoding allows future growth in a natural way. The coded file should be sharable by a number of applications – even if they cannot process everything that is in the file (Example: PIC, TIFF ([5])).

The main design goals of the PIC file format were extensibility and processing speed.

3.1 Block Structure

The key feature of PIC-Files is that they may be generated by different applications that support different atoms and compound objects. This implies that an application has to ignore all data not related to the internal set of supported objects. All information stored in a PIC-File is contained inside so-called *blocks* of the following form:

`ID`	`length`	$\underbrace{\texttt{data}}_{length}$	`[0] [0] [0]`

The `ID` is a 4-character (a 32-bit word) enumerate that identifies the type of the block. The `length` of the following data in bytes is also encoded as a 32-bit word. The meaning of the `data` following depends on the type of the block (the ID). In particular, it may contain other blocks, thus introducing a hierarchical file structure. Optional zero bytes (3 at maximum) are used to align blocks on 32-bit longword boundaries.
This block-oriented file structure has the following advantages:

a. The ID is a mnemonic name for the type of the block. When adding new blocks, name clashes have to be avoided. The length of the ID (4 characters) is a compromise between ease of adding new IDs and processing speed.

b. The PIC-Reader may easily skip over blocks with unknown IDs by using the length-of-data information. It is also possible to scan the file at high speed without interpreting the data.

c. The data field is fully transparent (any sequence of bytes may be stored here).

d. The smallest unit of processing is one byte. Blocks are longword-aligned, increasing the processing speed on most modern 32-bit processors.

e. Blocks can be nested. For instance, pictures may contain segments, the segments are composed of atoms. In addition, so-called *Property Blocks* may be used to define common properties of following blocks on the same level (Pascal-like scope rules).

The block ID is stored as a PIC enumerate (see section 4.2), the length is an unsigned 32-bit integer (see section 4.1.6). Throughout the remaining document, a block with ID `'ABCD'` will be referred to as `block'ABCD'`.

3.2 General Form of PIC Files

A PIC-File may contain more than one picture. The individual pictures are stored sequentially in the PIC-File, each of this form:

[block'CMNT']	block'PICT'

The optional `block'CMNT'` is a comment block that may contain anything and is ignored by the PIC reader. The `block'PICT'` contains the actual picture. Observe:

a. As stated in clause 3.1.b the PIC reader skips over encountered blocks of unknown type. The `block'PICT'` is currently the only defined block at the outer PIC file level that forces the PIC reader to inspect the block data. Therefore, any block other than the `block'PICT'` is ignored (like the `block'CMNT'`).

b. The `block'CMNT'`, however, is guaranteed to be ignored in future (extended) versions of PIC. Its use is to allow human-readable comments along with the pictures.

3.3 General Form of Pictures (Segments)

Pictures (segments) are stored in a `block'PICT'` containing the following data:

name	[block'FILE']	block'HEAD'	[block'PICT']*	block'BODY'

`Name` is a string of arbitrary length that holds the name of the segment. The string is terminated with one ore more zero bytes (to align the next block on a longword boundary), so the segment name may contain any character except the character `NUL`[1].

The optional `block'FILE'` is intended for use by the PIC Filing System. Typically, information such as names of segments referencing that segment, use counts or names of segments referenced by that segment are stored in this block. A File system may pass this information up to the PIC Reader or not. If the PIC Reader encounters a `block'FILE'`, it is ignored. Local segments contain no `block'FILE'`.

The `block'HEAD'` is the *Segment Header* and typically contains information concerning VDC extent, version, date and time of creation, etc. Values set in the segment header affect primitives stored in the corresponding picture body as well as in local segments defined in the corresponding `block'LSEG'` (unless overridden by the segment header of the local segment). The exact layout of the segment header is described in section 7.

A (possibly zero) number of `block'PICT'`s contain segments local to that segment (see figure 4). Values set in the `block'HEAD'` of the current picture (global segment) are automatically inherited into the local segments. The segment header of the local segment may redefine some of these values, not affecting other local segments or the segment body of the current picture (segment).

The `block'BODY'` is the *Segment Body*. It contains graphical primitives such as atoms (see section 5), compound objects (see section 6) or references to segments (see section 7.2).

In general, the structure of a PIC Segment is similar to the structure of a Pascal procedure containing constants and parameter declarations, declarations of procedures local to that procedure and the executable code of the procedure itself.

[1]This representation of strings has been chosen to be compatible with C and UNIX

4 Encoding of simple data types

This section describes the encoding of simple data types used in the encoding of more complex data (blocks). The following general rules apply:

a. The smallest unit of processing is one byte (8 bits). No information is stored in single bits or bitstrings. This increases file size, but speeds up processing (see goal 3.b on page 7).

b. The sizes of all simple data types are integer multiples of 8 bits.

c. Simple data types 32 bits long are aligned on longword (32 bit) boundaries. Simple data types 16 bits long are aligned on word (16 bit) boundaries. Single byte entities and strings of arbitrary length are aligned on byte boundaries. Again, this may increase file size by the need for pad bytes, but increases processing speed on most modern 32-bit processors.

d. Blocks are always aligned on longword boundaries (see clause 3.1.d on page 7).

e. The PIC file can be thought of as a stream of bytes. If a simple data type consists of more than one byte, the bytes are stored in decreasing order of significance. This corresponds to the byte order in Motorola 680x0-based computers. Unfortunately, the byte order in Intel 80x86-based systems is different, so conversion of words and longwords has to be performed in those PIC Readers and Writers.

4.1 Encoding of Integers

PIC supports signed and unsigned integers, each at three precisions: 8-bit, 16-bit and 32-bit. Signed integers are represented in 'two's complement' format.

4.1.1 Signed Integer at 8-bit precision

Each value occupies half a word (one byte). Numbers from -128 to +127 can be represented.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	msb			value i			lsb	S	msb		value i+1				lsb

4.1.2 Signed Integer at 16-bit precision

Each value occupies one word and may represent numbers from -32768 to +32767. This data type is always aligned at a word boundary within the PIC file.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	msb							value i							lsb

4.1.3 Signed Integer at 32-bit precision

Each value fills two consecutive words of the PIC file and may represent numbers from -2147483648 to +2147483647. This data type is always aligned at a longword boundary within the PIC file.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	msb								value i						
						value i									lsb

4.1.4 Unsigned Integer at 8-bit precision

Each value occupies half a word (one byte). Numbers from 0 to 255 may be represented. This data type is also used to represent a single character.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
msb			value i				lsb	msb			value i+1				lsb

4.1.5 Unsigned Integer at 16-bit precision

Each value occupies one word. Numbers from 0 to 65535 may be represented. This data type is always aligned at a word boundary within the PIC file.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
msb							value i								lsb

4.1.6 Unsigned Integer at 32-bit precision

Each value fills two consecutive words and may represent values from 0 to 4294967295. This data type is always aligned at a longword boundary within the PIC file.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
msb								value i							
						value i									lsb

4.2 Encoding of Enumerates

In order to gain flexibility of the format, PIC does not store enumerates as increasing integers. Instead, enumerates are stored as human-readable, 4-character names.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
char 1								char 2							
char 3								char 4							

The following rules for the choice of the name apply:

a. Enumerates should serve as a mnemonic name for a value. As far as this is possible with only 4 characters, they should be readable by humans.

b. Care has to be taken that no two applications use the same enumerate with different meanings.

c. If less than 4 characters are sufficient, the name should be left-aligned (`'DOT␣'`).

d. Characters should be considered case-sensitive. Only uppercase letters should be used for multi-application purposes, lowercase letters are reserved for private use.

e. Not only letters are allowed. `'_._.'` for a line style or `'LIN3'` for a line in 3D-space are reasonable names.

4.3 Encoding of Variable Names

Variable names are unique 32-bit numbers. It is application-dependent whether the variables are human-readable names (enumerates) or just unsigned integers (e.g. memory addresses). Therefore, the encoding of variable names is one of the following:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
char 1								char 2							
char 3								char 4							

or

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
msb								value i							
						value i									lsb

4.4 Encoding of Reals

Reals in PIC are binary floating point real values of 32-bit precision as standardized by the IEEE ([8]) and understood by most numeric coprocessors. This format contains three parts:

- a sign bit ('s')

- a biased exponent part ('e', 8 bits, bias=127)
- a fraction part ('f', 23 bits)

The value is a function of these three values. If 's' is '0', the value is positive; if 's' is '1', the value is negative. The magnitude of the value is calculated as follows:

a. If $e = 255$ and $f \neq 0$, then the value is undefined.

b. If $e = 255$ and $f = 0$, then the value is as large a positive ($s = 0$) or negative ($s = 1$) value as possible.

c. If $0 < e < 255$, then the magnitude of the value is $(1.f) \times (2^{e-127})$.

d. If $e = 0$ and $f \neq 0$, then the magnitude of the value is $(0.f) \times (2^{-126})$.

e. If $e = 0$ and $f = 0$, then the value is 0.

The approximate range of these 32-bit real values is

$$8.43 \times 10^{-37} \leq |X| \leq 3.37 \times 10^{38}$$

with approximately 7 significant decimal digits. Each real value occupies 2 consecutive words of the PIC file:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>S</td><td colspan="8">msb Exponent lsb</td><td colspan="7">msb Fraction</td></tr>
<tr><td colspan="16">Fraction lsb</td></tr>
</table>

4.5 Encoding of Flags

Flags occupy one byte of the PIC file. If all 8 bits of the flag byte are 0, the value of the flag is *false*, else (any bit set to 1) the value is *true*.

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="8">flag i</td><td colspan="8">flag i+1</td></tr>
</table>

4.6 Encoding of Strings

Strings are stored as an array of characters (8-bit unsigned integers). Strings may be of arbitrary length and are terminated by a NUL-character, as in this example:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="8">'S'</td><td colspan="8">'T'</td></tr>
<tr><td colspan="8">'R'</td><td colspan="8">'I'</td></tr>
<tr><td colspan="8">'N'</td><td colspan="8">'G'</td></tr>
<tr><td colspan="8">NUL</td><td colspan="8">value</td></tr>
</table>

Whether the `value` byte is used or not depends on the size of the next data item (see rule 4.c on page 9).

4.7 Encoding of Date and Time

The 'Date and Time' data type is encoded in UNIX-fashion: As an unsigned 32-bit integer giving the number of seconds since 00:00:00 on January 1, 1970.

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="16">msb Seconds since</td></tr>
<tr><td colspan="16">Jan 01,1970 00:00:00 lsb</td></tr>
</table>

4.8 Encoding of Colour

In order to allow maximum interchangeability of PIC files and to eliminate problems associated with colour lookup tables and segments, colour is encoded in direct RGB values.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
msb			reserved				lsb	msb			red				lsb
msb			green				lsb	msb			blue				lsb

The `reserved` byte is intended to allow future extension of the colour model and is currently set to 0. Note that colour lookup tables of arbitrary size may be software-simulated using the variable concept described in section 2.3.

4.9 Encoding of Coordinates

All Coordinates and Vectors are stored as two real values, the X and Y coordinates. Coordinates usually are subject to segment and workstation transformations (see section 2.4). To save space, encoding illustrations from now on are depicted using longwords (32 bits) instead of words (16 bits).

31 . . 24	23 . . 16	15 . . 8	7 . . 0
	Real:X		
	Real:Y		

4.10 Encoding of Linear Transformations

The transformation matrix used for linear 2D transformation of homogeneous coordinates looks like this:

$$\begin{bmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{bmatrix}$$

with components:

- a: x scale component
- b: x rotation component
- c: x translation component
- d: y rotation component
- e: y scale component
- f: y translation component

Obviously, only the (real) values a to f have to be stored:

31 . . 24	23 . . 16	15 . . 8	7 . . 0
	Real:a		
	Real:b		
	Real:c		
	Real:d		
	Real:e		
	Real:f		

5 Encoding of Atoms

This section defines the encoding of profile '2D' atoms of PIC. For a detailed description of the primitives and attributes refer to [10, part 3],[11, part 1] and [12, part 1].

Abbreviation	Simple Data Type
`byte`	Signed integer at 8-bit precision
`word`	Signed integer at 16-bit precision
`long`	Signed integer at 32-bit precision
`ubyte`	Unsigned integer at 8-bit precision
`uword`	Unsigned integer at 16-bit precision
`ulong`	Unsigned integer at 32-bit precision
`enum`	Enumerate
`var`	Variable name
`real`	Real number at 32-bit precision
`f`	Flag
`string`	String
`time`	Date and Time
`colour`	Colour
`coord`	Coordinate
`trans`	Transformation Matrix
`(pad)`	alignment pad (reserved, set to zero)

Table 2: *Abbreviations of simple data types*

Simple data types are abbreviated as shown in table 2. For instance, `var|ulong:length=5` means *variable or unsigned integer at 32-bit precision with meaning 'length' and value '5'*.

5.1 General Encoding of Atoms

31 . . 24 23 . . 16 15 . . 8 7 . . 0
`enum='ATOM'`
`ulong:length`
`enum:atomID`
`data`
.
.
.

Figure 5: *General Encoding of Atoms*

Atoms are stored as `block'ATOM'` (see figure 5) with the first data longword specifying the ID of the atom. As always, atoms of unknown ID may be ignored by the PIC Reader. Some atoms have variable parameter lists. In this case, the length field of the `block'ATOM'` (an unsigned 32-bit integer) is used to calculate the actual number of parameters (n). Note that `length` gives the size of the data plus 4 (the size of the enumerate specifying the atom ID) and that blocks are always aligned on longword boundaries.
For a list of atom IDs defined for profile '2D' refer to table 3.

5.2 Encoding of Attributes

In order to minimize the context sensitivity of the appearence of display primitives, PIC stores the attributes directly with the affected graphical primitives. To save space, the encoding of the attributes will be described only here and not for every atom.

ID	hex	Atom
`'ARC3'`	41524333	Circular Arc 3 Point
`'ARCB'`	41524342	Circular Arc Centre Backwards
`'ARCC'`	41524343	Circular Arc Centre
`'ARCE'`	41524345	Elliptical Arc
`'CA3C'`	43413343	Circular Arc 3 Point Close
`'CACC'`	43414343	Circular Arc Centre Close
`'CELL'`	43454C4C	Cell Array
`'CIRC'`	43495243	Circle
`'DISJ'`	4449534A	Disjoint Polyline
`'EAC '`	45414320	Elliptical Arc Close
`'ELPS'`	454C5453	Ellipse
`'LINE'`	4C494E45	Polyline
`'MARK'`	4D41524B	Polymarker
`'PIXL'`	5049584C	Pixel Array
`'POLY'`	504F4C59	Polygon
`'PSET'`	50534554	Polygon Set
`'RECT'`	52454354	Rectangle
`'RTXT'`	52545854	Simple Restricted Text
`'TEXT'`	54455854	Simple Text

Table 3: *Profile '2D' atom IDs in alphabetical order*

5.2.1 Encoding of Line Attributes

31 . . 24	23 . . 16	15 . . 8	7 . . 0
`f:TypeASF`	`f:WidthASF`	`f:ColourASF`	`(pad)`
`var|enum:LineType`			
`var|real:LineWidth`			
`var|colour:LineColour`			

Figure 6: *Encoding of Line Attributes*

Line Attributes are used by the atoms `'LINE'`, `'DISJ'`, `'ARC3'`, `'ARCC'`, `'ARCB'` and `'ARCE'`. They are encoded in a block of 4 longwords (16 bytes) as depicted in figure 6.

If `TypeASF` is `false`, `enum:Linetype` is the line type. Valid line types are shown in table 4. However, if `TypeASF` is `true`, `LineType` just stores the name of a variable that stores the line type or points to another variable, as described in section 2.3.

ID	hex	Line type
`'____'`	5F5F5F5F	solid
`'_ _ '`	5F205F20	dash
`'. . '`	2E202E20	dot
`'_._.'`	5F2E5F2E	dash-dot
`'_.. '`	5F2E2E20	dash-dot-dot

Table 4: *Line Types*

`WidthASF` and `ColourASF` perform a similar function for `LineWidth` and `LineColour`, respectively. In general, an ASF set true indicates that the corresponding attribute is bundled (i.e. a variable reference). An ASF set false indicates that the corresponding attribute is individual (i.e. specified directly).

If `WidthASF` is set to false, `Linewidth` stores the line width in VDC. A line width of zero means that the line should always be displayed using the minimum line width of a display device, otherwise the linewidth is subject to all transformations.
If `ColourASF` is set to false, `Linecolour` stores the line colour.

5.2.2 Encoding of Marker Attributes

31 . . 24	23 . . 16	15 . . 8	7 . . 0
f:TypeASF	f:SizeASF	f:ColourASF	(pad)
var|enum:MarkerType			
var|real:MarkerSize			
var|colour:MarkerColour			

Figure 7: *Encoding of Marker Attributes*

Marker Attributes are used only by the atom `'MARK'`. They are encoded in a block of 4 longwords (16 bytes) as shown in figure 7. The ASFs function as described for line attributes. Valid marker types are shown in table 5. The `MarkerSize` is given in VDC, `MarkerColour` stores the colour of the marker.

ID	hex	Marker Type
`'.   '`	2E202020	dot
`'+   '`	2B202020	plus
`'*   '`	2A202020	asterisk
`'O   '`	4F202020	circle
`'X   '`	58202020	cross

Table 5: *Marker Types*

5.2.3 Encoding of Fill Attributes

31 . . 24	23 . . 16	15 . . 8	7 . . 0
f:IStyleASF	f:FColourASF	f:HIndexASF	f:PIndexASF
f:FRefPointASF	f:PSizeASF	f:ETypeASF	f:EWidthASF
f:EColourASF	f:EdgeVisible	f:ImpEVisible	(pad)
var|enum:InteriorStyle			
var|colour:FillColour			
var|enum:HatchIndex			
var|enum:PatternIndex			
var|coord:FillReferencePoint			
var|coord:PatternSize.Height			
var|coord:PatternSize.Width			
var|enum:EdgeType			
var|real:EdgeWidth			
var|colour:EdgeColour			

Figure 8: *Encoding of Fill Attributes*

Fill Attributes are encoded in a block of 16 longwords (64 bytes) as shown in figure 8. They are used with the atoms `'POLY'`, `'PSET'`, `'RECT'`, `'CIRC'`, `'CA3C'`, `'CACC'`, `'ELPS'` and `'EAC'`.

Aspect Source Flag	Affected Attribute
`IStyleASF`	`InteriorStyle`
`FColourASF`	`FillColour`
`HIndexASF`	`HatchIndex`
`PIndexASF`	`PatternIndex`
`FRefPointASF`	`FillReferencePoint`
`PSizeASF`	`PatternSize.Height, PatternSize.Width`
`ETypeASF`	`EdgeType`
`EWidthASF`	`EdgeWidth`
`EColourASF`	`EdgeColour`

Table 6: *Aspect Source Flags of Fill Attributes*

A number of ASFs control the bundling of the Attributes (see table 6). Note:

- An ASF value of `false` (= 0) means *individual*, a value of `true` ($\neq 0$) means *bundled*.
- `EdgeVisible` and `ImpEVisible` are not ASFs, but control visibility of the explicit and implicit edges[2].
- Whereas PIC allows bundling of all attributes, CGM does not allow *Fill Reference Point, Pattern Size* and *Edge Visibility* to be bundled.
- In the case of `FRefPointASF=true` the variable name is stored in the first longword of `FillReferencePoint`, if `PSizeASF=true` two variable names are stored in the first longword of `Patternsize.Height` and in the first longword of `Patternsize.Width`.

Valid *Interior Styles* are shown in table 7 and valid *Hatch Indices* in table 8.

ID	hex	Interior Style
`'HOLO'`	`484F4C4F`	hollow
`'SOLD'`	`46494C4C`	solid
`'PTRN'`	`5054524E`	pattern
`'HTCH'`	`48544348`	hatch
`'EMPT'`	`454D5054`	empty

Table 7: *Interior Styles*

No *Pattern Index* is currently defined. *Edge Type, Edge Width* and *Edge Colour* are defined like *Line Type, Line Width* and *Line Colour*. *Fill Colour* is a colour value that defines the colour of the interior of the fill area.

The *Fill Reference Point* is given in VDC. The *Pattern Size* parameter consists of two VDC vectors (*Height, Width*) and defines the directions, in which the fill pattern shall be applied. The resolution of the pattern and the pattern itself is stored in a global pattern table. However, no patterns are defined for profile '2D'.

5.2.4 Encoding of Text Attributes

There are two kinds of text attributes:

[2] ImpEVisible is only meaningful with closed figures (see section 6.1).

<table>
<tr><th>ID</th><th>hex</th><th>Hatch Index</th></tr>
<tr><td>'----'</td><td>2D2D2D2D</td><td>horizontal</td></tr>
<tr><td>'||||'</td><td>7C7C7C7C</td><td>vertical</td></tr>
<tr><td>'////'</td><td>2F2F2F2F</td><td>positive slope</td></tr>
<tr><td>'\\\\'</td><td>5C5C5C5C</td><td>negative slope</td></tr>
<tr><td>'++++'</td><td>2B2B2B2B</td><td>combined vertical and horizontal slope</td></tr>
<tr><td>'XXXX'</td><td>58585858</td><td>combined left and right slant</td></tr>
</table>

Table 8: *Hatch Indices*

1. **Global Text Attributes** apply to simple texts (atoms `'TEXT'` and `'RTXT'`) and a whole compound text (compound objects `'TEXT'` and `'RTXT'`).
2. **Local Text Attributes** apply to simple and compound text elements and to individual substrings that form a compound text element (i.e. local text attributes may be changed within a compound text element).

<table>
<tr><th>31 . . 24</th><th>23 . . 16</th><th>15 . . 8</th><th>7 . . 0</th></tr>
<tr><td>f:TPrecASF</td><td>f:COrientASF</td><td>f:TPathASF</td><td>f:TAlignHASF</td></tr>
<tr><td>f:TAlignVASF</td><td>f:TAContHASF</td><td>f:TAContVASF</td><td>f:TPosASF</td></tr>
<tr><td colspan="4">var|enum:TextPrecision</td></tr>
<tr><td colspan="4">var|coord:CharacterOrientation.Up</td></tr>
<tr><td colspan="4">var|coord:CharacterOrientation.Base</td></tr>
<tr><td colspan="4">var|enum:TextPath</td></tr>
<tr><td colspan="4">var|enum:TextAlign.Horizontal</td></tr>
<tr><td colspan="4">var|enum:TextAlign.Vertical</td></tr>
<tr><td colspan="4">var|real:TextAlign.ContinuousHorizontal</td></tr>
<tr><td colspan="4">var|real:TextAlign.ContinuousVertical</td></tr>
<tr><td colspan="4">var|coord:TextPosition</td></tr>
</table>

Figure 9: *Encoding of Global Text Attributes*

Global Text Attributes are encoded in a block of 14 longwords (56 bytes) as shown in figure 9. They are used with the atoms `'TEXT'` and `'RTXT'` and the compound objects `'TEXT'` and `'RTXT'`.

<table>
<tr><th>31 . . 24</th><th>23 . . 16</th><th>15 . . 8</th><th>7 . . 0</th></tr>
<tr><td>f:TFontASF</td><td>f:CExpFactASF</td><td>f:CSpacingASF</td><td>f:TColourASF</td></tr>
<tr><td>f:CHeightASF</td><td>f:CSetASF</td><td>f:ACSetASF</td><td>(pad)</td></tr>
<tr><td colspan="4">var|enum:TextFontIndex</td></tr>
<tr><td colspan="4">var|real:CharacterExpansionFactor</td></tr>
<tr><td colspan="4">var|real:CharacterSpacing</td></tr>
<tr><td colspan="4">var|colour:TextColour</td></tr>
<tr><td colspan="4">var|real:CharacterHeight</td></tr>
<tr><td colspan="4">var|enum:CharacterSetIndex</td></tr>
<tr><td colspan="4">var|enum:AlternateCharacterSetIndex</td></tr>
</table>

Figure 10: *Encoding of Local Text Attributes*

Local Text Attributes are encoded in a block of 9 longwords (36 bytes) as shown in figure 10. They are used with the atoms `'TEXT'`, `'RTXT'` and the compound objects `'TEXT'` and `'RTXT'`. The block is always aligned on a longword boundary.

Aspect Source Flag	Affected Attribute
`TPrecASF`	`TextPrecision`
`COrientASF`	`CharacterOrientation.Up`, `CharacterOrientation.Base`
`TPathASF`	`TextPath`
`TAlignHASF`	`TextAlign.Horizontal`
`TAlignVASF`	`TextAlign.Vertical`
`TAContHASF`	`TextAlign.ContinuousHorizontal`
`TAContVASF`	`TextAlign.ContinouosVertical`
`TPosASF`	`TextPosition`
`TFontsASF`	`TextFontIndex`
`CExpFactASF`	`CharacterExpansionFactor`
`CSpacingASF`	`CharacterSpacing`
`TColourASF`	`TextColour`
`CHeightASF`	`CharacterHeight`
`CSetASF`	`CharacterSetIndex`
`ACSetASF`	`AlternateCharacterSetIndex`

Table 9: *Aspect Source Flags of Text Attributes*

A number of ASFs control the bundling of the Attributes (see table 9). Note:

- *TextPosition* is treated as a global attribute.
- Whereas PIC allows bundling of all attributes, CGM does not allow *Character Height, Character Orientation, Text Path, Text Alignment, Character Set Index, Alternate Character Set Index* and *TextPosition* to be bundled.
- In the case of `COrientASF=true` two variable names are stored in the first longword of `CharacterOrientation.Up` and in the first longword of `CharacterOrientation.Base`.
- If `TPosASF=true`, the variable name is stored in the first longword of `TextPosition`.

`TextFontIndex` specifies the name of the font to be used. Note that in CGM, `TextFontIndex` is only an index into a table of font names that is supplied in the Metafile Header. PIC directly specifies the font name with the text attributes so as to minimize context sensitivity. However, the appearence of fonts is implementation and application dependent. The only font defined for profile '2D' is `'SYS␣'`, the system-specific font every implementation provides.

ID	hex	Text Precision
`'STRI'`	`53545249`	string
`'CHAR'`	`43484152`	character
`'STRO'`	`5354524F`	stroke

Table 10: *Text Precisions*

Table 10 shows the available *Text Precisions*. *Character Expansion Factor, Character Spacing, Text Colour* and *Character Height* are defined as in CGM and CGI. The *Character Orientation* consists of two VDC vectors, *Up* and *Base*.

Table 11 shows the representations of the *Text Path*. The horizontal and vertical aligments are shown in tables 12 and 13, respectively.

ID	hex	Text Path
`'RGHT'`	`52474854`	right
`'LEFT'`	`4C454654`	left
`'UP  '`	`55502020`	up
`'DOWN'`	`444F574E`	down

Table 11: *Text Paths*

ID	hex	Horizontal Alignment
`'NORM'`	`4E4F524D`	normal horizontal
`'LEFT'`	`4C454654`	left
`'CNTR'`	`434E5452`	centre
`'RGHT'`	`52474854`	right
`'CONT'`	`434F4E54`	continuous horizontal

Table 12: *Horizontal Alignments*

Character Set Index is the mnemonic name of the character set that becomes the designated G0 set (by default invoked into positions hex 21 to hex 7E) for display of the Text. *Alternate Character Set Index* defines the designated G1 (by default invoked into positions hex A1 to hex FE) and also the designated G2 set. The use of these elements is to switch among character sets for different languages or block graphics. Note that in CGM, `CharacterSetIndex` and `AlternateCharacterSetIndex` are only indices to the *Character Set List* supplied in the Metafile Header. PIC directly specifies character sets to minimize context sensitivity. Profile '2D' defines only two 94-character G-sets: `'SYS0'`, the system-specific G0-set (characters, digits etc...) and `'SYS2'`, the system-specific G2-set (diacritical characters, ©, ß etc.) every implementation provides.

Control characters for character set invocation and designation (SI, SO, ESC, SS2 and SS3) are permitted in the text string. EDEN always assumes the *Extended 8-bit character coding* of CGM/CGI.

ID	hex	Vertical Alignment
`'NORM'`	`4E4F524D`	normal vertical
`'TOP '`	`544F5020`	top
`'CAP '`	`43415020`	cap
`'HALF'`	`48414C46`	half
`'BASE'`	`42415345`	base
`'BOTM'`	`424F544D`	bottom
`'CONT'`	`434F4E54`	continuous vertical

Table 13: *Vertical Alignments*

5.3 Encoding of Polyline and Disjoint Polyline

31 . . 24 23 . . 16 15 . . 8 7 . . 0
Header (figure 5) `length` = $8n + 20$ `atomID = 'LINE' | 'DISJ'`
Line Attributes (figure 6)
`coord:P1`
. . . .
`coord:P`n

Figure 11: *Encoding of Polyline and Disjoint Polyline*

The storage format of a Polyline or Disjoint Polyline is depicted in figure 11. After the header and line attributes, n coordinates of polyline vertices (for Polyline) or line segment endpoints (for Disjoint Polyline) follow. In case of a Disjoint Polyline, the number of points is even. For both atoms at least two points have to be supplied (see section 8 for the handling of error conditions). The value of n is determined by the `length` field of the `block'ATOM'` ($n = (length - 20)/8$).

5.4 Encoding of Circular Arc 3 Point

31 . . 24 23 . . 16 15 . . 8 7 . . 0
Header (figure 5) `length` = 44 `atomID = 'ARC3'`
Line Attributes (figure 6)
`coord:StartingPoint`
`coord:IntermediatePoint`
`coord:EndingPoint`

Figure 12: *Encoding of Circular Arc 3 Point*

Circular Arc 3 Point is stored in a block of 11 longwords (44 bytes, including the atom ID) as depicted in figure 12.

5.5 Encoding of Circular Arc Centre and Circular Arc Centre Backwards

Circular Arc Centre and *Circular Arc Centre Backwards* atoms are stored in a block of 12 longwords (48 bytes, including the atom ID) as depicted in figure 13. The atoms differ only in their atom ID. The Radius is given in VDC.

31 . . 24	23 . . 16	15 . . 8	7 . . 0
Header (figure 5) `length` = 48 `atomID = 'ARCC' | 'ARCB'`			
Line Attributes (figure 6)			
`coord:Centre`			
`coord:StartVector`			
`coord:EndVector`			
`real:Radius`			

Figure 13: *Encoding of Circular Arc Centre and Circular Arc Centre Backwards*

5.6 Encoding of Elliptical Arc

31 . . 24	23 . . 16	15 . . 8	7 . . 0
Header (figure 5) `length` = 60 `atomID = 'ARCE'`			
Line Attributes (figure 6)			
`coord:Centre`			
`coord:CD1`			
`coord:CD2`			
`coord:StartVector`			
`coord:EndVector`			

Figure 14: *Encoding of Elliptical Arc*

An *Elliptical Arc* is stored in a block of 15 longwords (60 bytes, including the atom ID) as depicted in figure 14. `CD1` and `CD2` are the endpoints of the conjugate diameters.

5.7 Encoding of Polymarker

31 . . 24	23 . . 16	15 . . 8	7 . . 0
Header (figure 5) `length` = $8n + 20$ `atomID = 'MARK'`			
Marker Attributes (figure 7)			
`coord:P1`			
. . . .			
`coord:P`n			

Figure 15: *Encoding of Polymarker*

The storage format of a Polymarker is depicted in figure 15. After the header and marker attributes, n coordinates of polymarkers follow. As with polyline and disjoint polyline, the **`length`** field of the **`block'ATOM'`** is used to determine the number of points ($n = (length - 20)/8$).

5.8 Encoding of Polygon

31 . . 24	23 . . 16	15 . . 8	7 . . 0
Header (figure 5) `length` = $8n + 68$ `atomID = 'POLY'`			
Fill Attributes (figure 8)			
`coord:P1`			
. . . .			
`coord:P`n			

Figure 16: *Encoding of Polygon*

The storage format of a Polygon is depicted in figure 16. After the header and fill attributes, $n \geq 3$ edge points follow. n is calculated from the **`length`** field of the **`block'ATOM'`** ($n = (length - 68)/8$).

5.9 Encoding of Polygon Set

The storage format of a Polygon Set is depicted in figure 17. After the header and fill attributes $n \geq 3$ edge points follow. In addition to the point coordinates, every edge point also stores two flags,

31 . . 24	23 . . 16	15 . . 8	7 . . 0
Header (figure 5) `length` = $12n + 68$ `atomID` = 'PSET'			
Fill Attributes (figure 8)			
`f:Visible1`	`f:Close1`	(pad)	(pad)
`coord:P1`			
. . . .			
`f:Visible`n	`f:Close`n	(pad)	(pad)
`coord:P`n			

Figure 17: *Encoding of Polygon Set*

the *Edge Visibility Flag* and the *Closure Flag*, both valid for the outgoing edge. n is calculated from the **length** field of the **block'ATOM'** ($n = (length - 68)/12$).

5.10 Encoding of Rectangle

31 . . 24	23 . . 16	15 . . 8	7 . . 0
Header (figure 5) `length` = 84 `atomID` = 'RECT'			
Fill Attributes (figure 8)			
`coord:P1`			
`coord:P2`			

Figure 18: *Encoding of Rectangle*

A Rectangle and the associated attributes are stored in a block of 21 longwords (84 bytes, including the atom ID) as depicted in figure 18. Note: the corners of the rectangle (P1, P2) are not sorted in any way.

5.11 Encoding of Circle

A Circle and the associated attributes are stored in a block of 20 longwords (80 bytes, including the atom ID) as depicted in figure 19. The radius is given in VDC.

5.12 Encoding of Circular Arc 3 Point Close

A Circular Arc 3 Point Close and the associated attributes are stored in a block of 24 longwords (96 bytes, including the atom ID) as shown in figure 20. The valid *Closure Types* are shown in table 14.

31 . . 24 23 . . 16 15 . . 8 7 . . 0

Header (figure 5)
`length = 80`
`atomID = 'CIRC'`

Fill Attributes (figure 8)
`coord:Centre`
`real:Radius`

Figure 19: *Encoding of Circle*

31 . . 24 23 . . 16 15 . . 8 7 . . 0

Header (figure 5)
`length = 96`
`atomID = 'CA3C'`

Fill Attributes (figure 8)
`enum:ClosureType`
`coord:StartingPoint`
`coord:IntermediatePoint`
`coord:EndingPoint`

Figure 20: *Encoding of Circular Arc 3 Point Close*

5.13 Encoding of Circular Arc Centre Close

A Circular Arc Centre Close and the associated attributes are stored in a block of 25 longwords (100 bytes, including the atom ID) as shown in figure 21. The valid *Closure Types* are the same as for *Circular Arc 3 Point Close* and shown in table 14.

5.14 Encoding of Ellipse

The Ellipse and the associated attributes are stored in a block of 23 longwords (92 bytes, including the atom ID) as shown in figure 22. CD1 and CD2 denote the end points of a conjugate diameter pair.

5.15 Encoding of Elliptical Arc Close

The Elliptical Arc Close and the associated attributes are stored in a block of 28 longwords

ID	hex	Closure Type
'PIE '	50494520	pie closure
'CHRD'	43485244	chord closure

Table 14: *Arc Closure Types*

31 . . 24 23 . . 16 15 . . 8 7 . . 0
Header (figure 5) `length` = 100 `atomID = 'CACC'`
Fill Attributes (figure 8)
`enum:ClosureType`
`coord:Centre`
`coord:StartVector`
`coord:EndVector`
`real:Radius`

Figure 21: *Encoding of Circular Arc Centre Close*

31 . . 24 23 . . 16 15 . . 8 7 . . 0
Header (figure 5) `length` = 92 `atomID = 'ELPS'`
Fill Attributes (figure 8)
`coord:Centre`
`coord:CD1`
`coord:CD2`

Figure 22: *Encoding of Ellipse*

(112 bytes, including the atom ID) as shown in figure 23. The valid *Closure Types* are the same as for *Circular Arc 3 Point Close* and shown in table 14. CD1 and CD2 denote the end points of the conjugate diameters.

5.16 Encoding of Simple Text

The storage format of Simple Text is shown in figure 24. Only a single text string can be stored as a Simple Text. If attribute changes within the text string are needed, Compound Text (see section 6.2) is to be used. Simple Text corresponds to the *Text* primitive of CGM/CGI with the *Final Flag* set to *final*.

The length of the string (n), may be found by using the `length` of the `block'ATOM'` or by triggering on the terminating `NUL`-byte. Note that n is the length of the string *including* the `NUL`-byte.

5.17 Encoding of Simple Restricted Text

The storage format of Simple Restricted Text is shown in figure 25. Only a single text string can be stored as a Simple Restricted Text. If attribute changes within the text string are needed, Compound Restricted Text (see section 6.3) is to be used. Simple Restricted Text corresponds to

31 . . 24	23 . . 16	15 . . 8	7 . . 0

Header (figure 5)
`length = 112`
`atomID = 'EAC '`			

Fill Attributes (figure 8)			
`enum:ClosureType`			
`coord:Centre`			
`coord:CD1`			
`coord:CD2`			
`coord:StartVector`			
`coord:EndVector`			

Figure 23: *Encoding of Elliptical Arc Close*

31 . . 24	23 . . 16	15 . . 8	7 . . 0
Header (figure 5) `length = 92+`n `atomID = 'TEXT'`			
Global Text Attributes (figure 9)			
Local Text Attributes (figure 10)			
`string:Text`			

Figure 24: *Encoding of Simple Text*

the *Restricted Text* primitive of CGM/CGI with the *Final Flag* set to *final*.

The length of the string (n), may be found by using the `length` of the `block'ATOM'` or by triggering on the terminating `NUL`-byte. Note that n is the length of the string *including* the `NUL`-byte.

5.18 Encoding of Cell Array

The encoding of *Cell Array* is shown in figure 26. `P, Q` and `R` are three corners of a parallelogram (the fourth is computed). `nx` is the number of cells in direction $\overline{PR}$, `ny` the number of cells in direction $\overline{RQ}$. `Compression` is the compression mode used for storing the colour list (see table 15).

If `Compression` is set to `'NONE'`, the colours of the cells are simply stored as `colour` data types, in rows parallel to $\overline{PR}$, starting at P.

The `'PACK'` Compression Mode is described in Appendix B. If an unknown Compression Mode is encountered, the preferred action of the application is to ignore the data and just draw the

31 . . 24	23 . . 16	15 . . 8	7 . . 0
Header (figure 5) length = 100+n atomID = 'RTXT'			
real:ExtentWidth			
real:ExtentHeight			
Global Text Attributes (figure 9)			
Local Text Attributes (figure 10)			
string:Text			

Figure 25: *Encoding of Simple Restricted Text*

31 . . 24	23 . . 16	15 . . 8	7 . . 0
Header (figure 5) length = 36+length(data) atomID = 'CELL'			
coord:P			
coord:Q			
coord:R			
uword:nx		uword:ny	
enum:Compression			
data . . .			

Figure 26: *Encoding of Cell Array*

outline of the parallelogram.

5.19 Encoding of Pixel Array

The encoding of *Pixel Array* is shown in figure 27. `Origin` is the location in VDC space at which the first colour value is to be placed. `nx` is a signed integer, the absolute value of which

ID	hex	Compression Mode
'NONE'	4E4F4E45	no compression
'PACK'	5041434B	packed run length

Table 15: *Cell and Pixel Array Compression Modes*

31 . . 24	23 . . 16	15 . . 8	7 . . 0

Header (figure 5)
`length = 32+length(data)`
`atomID = 'PIXL'`			

`coord:Origin`			
`word:nx`		`word:ny`	
`word:xmin`		`word:xmax`	
`word:ymin`		`word:ymax`	
`uword:xscale`		`uword:yscale`	
`enum:Compression`			
`data` . . .			

Figure 27: *Encoding of Pixel Array*

defines how many pixels are in each row. If **`nx`** is positive, the row extends toward increasing x from the origin point. If **`nx`** is negative, it extends toward decreasing x from the origin point. The corresponding is true for **`ny`**.

`xmin..xmax` and **`ymin..ymax`** specify the subrectangle within the array which is to be drawn. If the subrectangle extends beyond the limits of the **`nx`** by **`ny`** array, only the common part of the two rectangles is to be rendered.

`xscale` and **`yscale`** permit integer scaling of the pixel array independently in x and y dimensions. These scale factors are positive integers.

Colour values are compressed like the colours of cell array (see table 15).

5.20 Other Atoms

The set of atoms can easily be extended. For example, a spline curve needed for some applications (but not part of CGM/CGI and therefore not part of PIC profile '2D') could be defined using the same storage layout as a (disjoint) polyline. Only a unique atom ID for the spline has to be defined, e.g. `'SPL2'`. Also, a profile extending the set of atoms has to specify whether these new atoms may be part of a closed figure or part of a compound text (see sections 6.1 and 6.2).

6 Encoding of Compound Objects

31 . . 24	23 . . 16	15 . . 8	7 . . 0
enum='COMP'			
ulong:length			
enum:CompoundID			
data . . .			

Figure 28: *General Encoding of Compound Objects*

A compound object is stored as `block'COMP'` (see figure 28), very much like atoms. The first data longword specifies the ID of the compound object. `CompoundIDs` defined for profile '2D' are listed in table 16. As always, compound objects of unknown `CompoundID` are ignored by the PIC Reader. Note that `length` gives the size of the data plus 4 (the size of the enumerate specifying the compound ID) and that blocks are always aligned on longword boundaries.

ID	hex	Compound Object
'CFIG'	43464947	Closed Figure
'RTXT'	52545854	Compound Restricted Text
'TEXT'	54455854	Compound Text

Table 16: *Profile '2D' CompoundIDs in alphabetical order*

6.1 Encoding of Closed Figure

31 . . 24	23 . . 16	15 . . 8	7 . . 0
Header (figure 28) length = ? CompoundID = 'CFIG'			
Fill Attributes (figure 8)			
Segment Reference (figure 37)			

Figure 29: *Encoding of Closed Figure*

The storage format of a *Closed Figure* is shown in figure 29. After the fill attributes a Segment Reference (a `block'LSEG'` or `block'GSEG'`, see section 7.2) follows. This segment stores the graphic primitives the closed figure consists of. During the display process, these primitives are rendered with respect to the following rules:

1. If the segment itself contains other segments, these sub-segments are treated as individual regions of the closed figure, realizing the *New Region* concept of CGM/CGI.
2. Any compound objects (including closed figures) in the segment or sub-segments are ignored.
3. Atoms found in table 17 cannot be part of a closed figure and are ignored[3].

[3] This table will have to be extended if new atoms are defined.

ID	Atom
`'CELL'`	Cell Array
`'MARK'`	Polymarker
`'PIXL'`	Pixel Array
`'RTXT'`	Restricted Text
`'TEXT'`	Text

Table 17: *Atoms that cannot be part of a closed figure*

4. Other atoms (line and fill primitives) are rendered and the interior is filled using the given fill attributes.
5. The edge attributes *Edge Type* and *Edge Width* are local attributes, i.e. within the closed figure, line or fill primitives may specify their own (different) values for these attributes. *Edge Visibilty* is also a local attribute and applies only to explicitly specified edges (see 7).
6. All other attributes are global attributes, i.e. they are applied as specified in the closed figure and the values associated with the atoms comprising the closed figure are ignored.
7. *Implicit Edge Visibility* is also a global attribute and affects the visibility of implicit edges. In the construction of closed figures, implicit edges are generated:
   (a) between the last point of one line primitive and the first point of the next, if the two points do not coincide.
   (b) between the last point of a line primitive and the current closure point, if the two points do not coincide, when a new region is started. A new region is started by a segment reference or a fill primitive. The current closure point is the first point of the first primitive of a region.
   (c) between all points $(2i + 2)$ and $(2i + 3), i \geq 0$ of a disjoint polyline used to define a boundary.

In addition, all the features of segments apply:

- The atoms of the segment are subject to 2D transformation.
- The segment may contain sub-segments, the atoms of which form regions of the closed figure.
- The segment may be parametrized.
- The segment name itself may be a parameter.

This way, the user may create a commonly used shape (e.g. a rectangle with rounded corners) as a segment once and for all, and fill it (or not) with arbitrary fill attributes by referencing it in a closed figure. Segment transformation allows to display this shape in any desired size and orientation.

6.2 Encoding of Compound Text

31 . . 24	23 . . 16	15 . . 8	7 . . 0

Header (figure 28)
`length = ?`
`CompoundID = 'TEXT'`			

Global Text Attributes (figure 9)			
Local Text Attributes (figure 10)			
`string:Text` . . .			
. . . .			
Local Text Attributes (figure 10)			
`string:Text` . . .			

Figure 30: *Encoding of Compound Text*

The storage format of Compound Text is shown in figure 30. The global text attributes apply to the whole compound object, while the local text attributes are given for every substring. Compound Text corresponds to the *Text* primitive of CGM/CGI with the *Final Flag* set to *not final* and subsequent *Append Text* primitives.

After the text strings a number of pad bytes are inserted to align the next Local Text Attribute block on a longword boundary. The end of the strings can only be found by triggering on the terminating `NUL`-byte.

6.3 Encoding of Compound Restricted Text

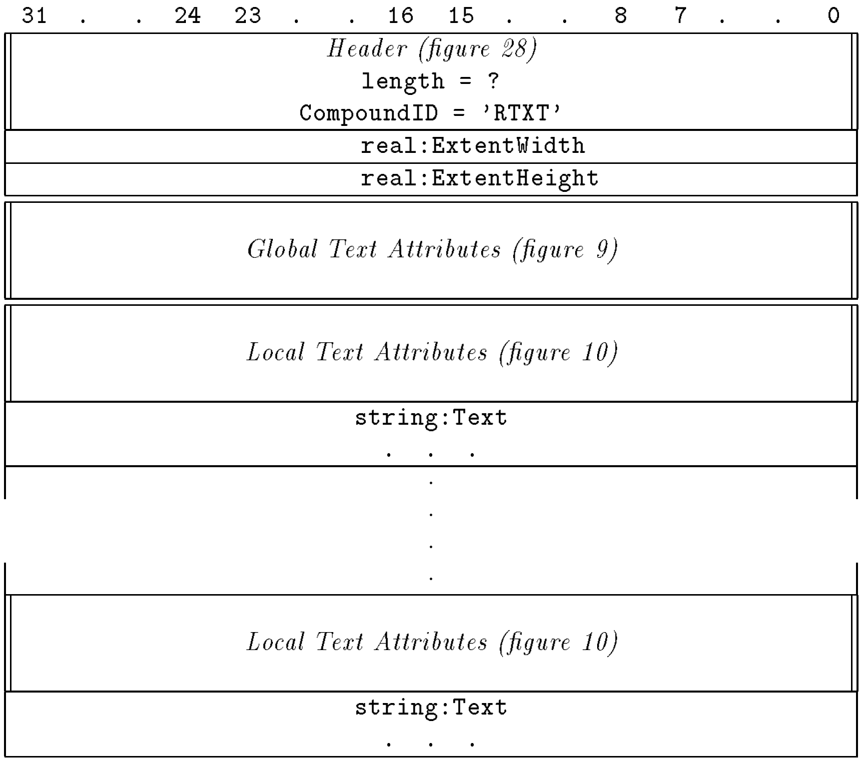

Figure 31: *Encoding of Compound Restricted Text*

The storage format of Compound Restricted Text is shown in figure 31. The *ExtentWidth, ExtentHeight* and global text attributes apply to the whole compound object, while the local text attributes are given for every substring. Compound Restricted Text corresponds to the *Restricted Text* primitive of CGM/CGI with the *Final Flag* set to *not final* and subsequent *Append Text* primitives.

After the text strings a number of pad bytes are inserted to align the next Local Text Attribute block on a longword boundary. The end of the strings can only be found by triggering on the terminating `NUL`-byte.

7 Encoding of Segments

As outlined in section 3.3, segments are stored just like pictures, i.e. pictures <u>are</u> segments. There are, however, global and local segments. While there is no substantial difference in the storage layout of global versus local segments, the segment reference is different. In addition, the use of local instead of global segments reduces file system overhead and increases processing speed.

31 . . 24 23 . . 16 15 . . 8 7 . . 0
`enum:blockID='PICT'`
`ulong:length`
`string:Name` . . .
`[block'FILE']`
`block'HEAD'`
`[block'PICT']*`
`block'BODY'`

Figure 32: *Encoding of Segments (Pictures)*

Segments (global or local) are stored in a `block'PICT'` as illustrated in figure 32. `Name` is a string of arbitrary length that holds the name of the segment. The string is terminated with one ore more zero bytes (to align the next block on a longword boundary), so the segment name may contain any character except for the character `NUL`. Segment names are case-sensitive.

The optional `block'FILE'` is intended for use by the PIC Filing System (see section 10). Typically, information such as names of segments referencing that segment, use counts or names of segments referenced by that segment are stored in this block. A File system may pass this information up to the PIC Reader or not. Specification of the `block'FILE'` storage layout is not subject to this documentation. If the PIC Reader encounters a `block'FILE'`, it is ignored. Local segments usually contain no `block'FILE'`.

The `block'HEAD'` is the *Segment Header* described in section 7.1. Values set in the segment header affect primitives stored in the corresponding segment body as well as in local segments defined in the corresponding `block'LSEG'` (unless overridden by the segment header of the local segment).

A (possibly zero) number of `block'PICT'`s contain segments local to that segment (note that the definition of `block'PICT'` is recursive). Values set in the `block'HEAD'` of the current segment are inherited to the local segments. The segment header of the local segment may redefine some of these values, not affecting other local segments or the segment body of the current segment.

The `block'BODY'` is the *Segment Body*. It contains graphical primitives such as atoms (see section 5), compound objects (see section 6) or references to segments (see section 7.2), i.e. a sequence of `block'ATOM'`s, `block'COMP'`s, `block'LSEG'`s and `block'GSEG'`s.

In contrast to CGM/CGI, no explicit *Segment Priority* is defined in PIC. However, display order implies a priority relation among objects (atoms, compound objects and segments) according to the *Painter's algorithm*: The segment with highest priority is displayed last. PIC reflects this priority scheme by defining objects in the `block'BODY'` in display order, which means that objects are sorted in ascending display priority inside the `block'BODY'`.

7.1 Encoding of the Segment Header

31 . . 24 23 . . 16 15 . . 8 7 . . 0
enum:blockID='HEAD'
ulong:length
[block'TIME']*
[block'VDCX']
[block'PARM']*
[block'SET ']*

Figure 33: *Encoding of the Segment Header*

The segment header contains a number of segment properties coded as *Segment Property Blocks* (see figure 33). These blocks are described in the following sections.

7.1.1 Encoding of Segment Property Block 'TIME'

31 . . 24 23 . . 16 15 . . 8 7 . . 0
enum:blockID='TIME'
ulong:length
real:Version
time:DateAndTime
string:Application . . .
string:Author . . .
string:Machine . . .
string:Comment . . .

Figure 34: *Encoding of Segment Property Block 'TIME'*

The encoding of `block'TIME'` is shown in figure 34. Note:

- Multiple `block'TIME'`s are allowed in a single `block'HEAD'` as long as they are distinguishable by their `Version` numbers. The EDEN application writing the file may choose to:
  - Write no `block'TIME'`. This option should not be used for global segments. Local segments of minor significance may be stored with no `block'TIME'` at all, as the version info is stored with the referencing global segment anyway.
  - Write a single `block'TIME'` corresponding to the last modification of the segment.
  - Write two `block'TIME'`s corresponding to first creation and last modification of the segment. The `block'TIME'` with higher version number reflects last modification while the version number of first creation is usually 1.0.

  – Write a new `block'TIME'` at every modification, not discarding the others. This way the complete modification history of a segment is recorded and may be retrieved by sorting the stored `block'TIME'`s by `Version` or `DateAndTime`.

- `Version` numbers are stored as reals, allowing major and minor version numbers (e.g. version 1.25). Version numbers smaller than 1.0 should not be used.
- `DateAndTime` is stored in UNIX-compatible fashion (see section 4.7) and reflects the Date and Time of segment creation/modification in GMT.
- The `Application` string tells the name of the program – usually an EDEN-based graphics editor or a code converter (see figure 1) – that created/modified the segment (e.g. 'BTX/CEPT V1.2').
- `Author` holds the name of the person that created/modified the segment. If the author is known by real-world name, it is stored here. If only the login name is available, that name is used. If there is no author (e.g. segment is created by some decoder process), the user id of the process owner should be saved.
- `Machine` keeps the name of the machine on which the author created/modified the segment, preferably a logical node name like the one used for E-mail (or else the name of the organization and a machine name unique in that organization).
- `Comment` may be used to store an arbitrarily long comment on the purpose of the segment, limitations, reason for modification etc. The string is intended to be interactively entered by the user and is not supposed to carry application-specific information (application-specific property blocks may be used for that purpose).
- All strings are aligned on byte boundaries. However, after the `Comment` string a sufficient number of pad bytes is inserted to align the next block on a longword boundary.
- As strings are terminated by a `NUL`-byte, they may contain all other characters and are of arbitrary length. The empty string is a single `NUL`-byte.

7.1.2 Encoding of Segment Property Block 'VDCX'

31 . . 24 23 . . 16 15 . . 8 7 . . 0
enum:blockID='VDCX'
ulong:length=16
real:MinX
real:MaxX
real:MinY
real:MaxY

Figure 35: *Encoding of Segment Property Block 'VDCX'*

The *VDC Extent* (see figure 35) specifies the range of coordinates in the segment. Specification of coordinates outside the VDC extent is permitted. It is intended that the visible portion of the segment be contained within the VDC extent. For example, an EDEN application may display the bounding rectangle of a segment by using the VDC extent of the segment header.

An application may choose to describe pictures in coordinates that map directly to the Device Coordinates (DC) of a target device, but still are displayed correctly on other devices. Whether the mapping of VDC to DC is isotropic or not is implementation-dependent.

Only one `block'VDCX'` is allowed per segment header. If the `block'VDCX'` is missing, the default VDC extent is used (see section 8).

7.1.3 Encoding of Segment Property Block 'SET '

The property `block'SET '` (see figure 36) is used to set a variable (see section 2.3) to a value of a certain type.

31 . . 24 23 . . 16 15 . . 8 7 . . 0
`enum:blockID='SET '|'PARM'`
`ulong:length=8+length(value)`
`var:Name`
`enum:Type`
`Type:Value` ...

Figure 36: *Encoding of Segment Property Blocks 'SET ' and 'PARM'*

`Name` is the target variable name. `Type` indicates the type of the variable, according to table 18.

ID	hex	Atom
'SI8 '	53493820	Signed integer at 8-bit precision
'SI16'	53493136	Signed integer at 16-bit precision
'SI32'	53493332	Signed integer at 32-bit precision
'UI8 '	55493820	Unsigned integer at 8-bit precision
'UI16'	55493136	Unsigned integer at 16-bit precision
'UI32'	55493332	Unsigned integer at 32-bit precision
'ENUM'	454E554D	Enumerate
'VAR '	56415220	Variable name
'REAL'	5245414C	Real number at 32-bit precision
'FLAG'	464C4147	Flag
'STRI'	53545249	String
'COLO'	434F4C4F	Colour
'COOR'	434F4F52	Coordinate
'TMAT'	544D4154	Transformation Matrix

Table 18: *Profile '2D' data types*

If `Type` is not `'VAR '`, the actual value is stored in `Value`. `Value` is a value of the indicated type. The length of `Value` may be determined by the `length` field of the `block'SET '`.
If `Type` is `'VAR '`, `Value` stores the source variable name. Note:

- The evaluation of variables is not performed by the PIC Reader but by the display process.
- Wherever a variable is supposed to store some value (e.g. attributes, segment name) the variable may also just hold the name of another variable, that eventually stores the desired value.
- As the source variable may also be set to another variable, variables may be indirected a number of times. This feature is used to pass parameters from upper-level segments down to lower-level segments (see section 7.2).
- The value set by a `block'SET '` is only visible in the corresponding segment body and local segments defined in the segment containing the `block'SET '`. A local segment may override this value in its segment header, not affecting the other segments of same or higher level or the segment body of the current segment (Pascal-like scope rules).

The segment header may contain an arbitrary number of `block'SET '`s.

7.1.4 Encoding of Segment Property Block 'PARM'

The property `block'PARM'` is encoded like the `block'SET '` (see figure 36), but with a slightly different meaning. This block is used to declare the names of variables that are parameters of the segment, i.e. variables that may be used within the segment and assigned values to from outside the segment (by segment reference).

`Name` is the name, `Type` is the data type of the variable (=parameter). `Value` may be used to assign a default value to the parameter. The use of `Value` is optional (if not used, `length`=8).

The segment header may contain an arbitrary number of `block'PARM'`s.

The segment header of PIC might be compared to the procedure declaration part of a block-structured programming language (e.g. Pascal). The 'PARM' blocks represent the declaration of formal parameters[4], while the 'SET ' blocks correspond to the declarations of local constants.

7.2 Encoding of Segment Reference

There are in fact two different segment reference blocks:

1. `block'GSEG'` references a global segment. Usually the system will have to fetch the segment from secondary storage, e.g. the PIC Filing System (see section 10). There is a single name space for global segments, i.e. two different global segments must not have the same name.
2. `block'LSEG'` references a local segment. Usually the system manages to keep local segments in primary storage. To find the desired segment, the segment names defined in the `block'PICT'`s of the current segment are scanned first. If the segment is not found there, the `block'PICT'` with the definition of the current segment is searched (as far as it is read in), and then the way up to the root segment, very much like PASCAL scope rules.

<table>
<tr><th>31 . . 24</th><th>23 . . 16</th><th>15 . . 8</th><th>7 . . 0</th></tr>
<tr><td colspan="4">enum:blockID='LSEG'|'GSEG'</td></tr>
<tr><td colspan="4">ulong:length</td></tr>
<tr><td>f:NameASF</td><td>f:TransASF</td><td colspan="2">(pad)</td></tr>
<tr><td colspan="4">var|string:Name
...</td></tr>
<tr><td colspan="4">trans:TMatrix</td></tr>
<tr><td colspan="4">[block'SET ']*</td></tr>
</table>

Figure 37: *Encoding of Segment Reference*

This distinction helps not only the system to find the segment, but – more important – the user of the system to define segments:

- When defining a global segment, the user does not need to care about existing local segments with the same name.
- When defining a local segment within a segment, no name clashes with global segments can occur.
- Also, name clashes with local segments defined within other global or local segments are resolved by the system.

[4] It is not possible to assign default values to formal parameters in Pascal, but it is in C++ [16], for example.

- When incorporating a previously defined segment, possible name clashes with other segments already defined are automatically resolved.

The storage layout of `block'LSEG'` and `block'GSEG'` is shown in figure 37. If `NameASF=false`, the segment name is stored directly in `Name` and no pad bytes are inserted before `Name`. If `NameASF=true`, however, the next longword stores the name of a variable, that stores the segment name. In conjunction with `block'SET '` and `block'PARM'`, this feature allows one to parametrize the name of a segment some other segment is to reference, just like passing function names as parameters to functions in a programming language.

`TMatrix` is the 2D transformation matrix (see section 4.10) that is to be applied to the referenced segment. `TransASF` allows bundling of the transformation matrix. If `TransASF=true`, the first longword of `Tmatrix` (which is the x scale component, a) stores the name of the variable that contains the transformation matrix.

Zero or more `block'SET '`s are part of the segment reference and set formal parameters of the referenced segment to actual values (or to the values of other variables or parameters).