Palm™ File Format Specification
# Table of Contents

## About This Document

## 1 PQA Database Format

- Overview .................................................. 1
- PQA Database Format ..................................... 2
- PDB Header for a PQA .................................... 3
- Record List in the PDB Header ......................... 7
- PQA appInfo Block ....................................... 8
- Web Content Record ..................................... 11

## 2 Content And Compression Types

- cmlContentTypeTextCml .................................. 18
- Plain Text .................................................. 18
- cmlContentTypeImagePalmOS ............................ 18
- cmlCompressionTypeNone and cmlCompressionTypeBitPacked 18

## 3 PQA Encoding Format

- Overview .................................................. 21
- Differences From HTML ................................. 23
- Bit Packed Compression ................................. 24
  - Bit Packed Compression Encoding .................. 28
  - ASCII Text Encoding ................................. 28
  - Tag Encoding ........................................ 28
  - Numeric Parameter Value Encoding ............... 29
  - Image Compression .................................. 29
- Compact Data Structure Notation ..................... 29
- Compact Data Structure Types ....................... 30
  - UIntV .................................................. 30
  - IntV .................................................. 31
  - Uint16V ............................................... 31
  - Int16V ............................................... 31
  - Uint8V ............................................... 31
<table>
<thead>
<tr>
<th>Tag Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmlTagListDefinition</td>
<td>51</td>
</tr>
<tr>
<td>cmlTagListItemNormal</td>
<td>52</td>
</tr>
<tr>
<td>cmlTagListItemCustom</td>
<td>52</td>
</tr>
<tr>
<td>cmlTagListItemTerm</td>
<td>54</td>
</tr>
<tr>
<td>cmlTagListItemDefinition</td>
<td>54</td>
</tr>
<tr>
<td>Forms</td>
<td>54</td>
</tr>
<tr>
<td>cmlTagForm</td>
<td>54</td>
</tr>
<tr>
<td>cmlTagInputTextLine</td>
<td>57</td>
</tr>
<tr>
<td>cmlTagInputPassword</td>
<td>58</td>
</tr>
<tr>
<td>cmlTagInputRadio</td>
<td>59</td>
</tr>
<tr>
<td>cmlTagInputCheckBox</td>
<td>60</td>
</tr>
<tr>
<td>cmlTagInputSubmit</td>
<td>61</td>
</tr>
<tr>
<td>cmlTagInputReset</td>
<td>62</td>
</tr>
<tr>
<td>cmlTagInputHidden</td>
<td>63</td>
</tr>
<tr>
<td>cmlTagInputTextArea</td>
<td>63</td>
</tr>
<tr>
<td>cmlTagSelect</td>
<td>64</td>
</tr>
<tr>
<td>cmlTagSelectItemNormal</td>
<td>65</td>
</tr>
<tr>
<td>cmlTagSelectItemCustom</td>
<td>65</td>
</tr>
<tr>
<td>cmlTagInputDatePicker</td>
<td>66</td>
</tr>
<tr>
<td>cmlTagInputTimePicker</td>
<td>67</td>
</tr>
<tr>
<td>Tables</td>
<td>67</td>
</tr>
<tr>
<td>cmlTagTable</td>
<td>67</td>
</tr>
<tr>
<td>cmlTagCaption</td>
<td>69</td>
</tr>
<tr>
<td>cmlTagTableRow</td>
<td>70</td>
</tr>
<tr>
<td>cmlTagTableData</td>
<td>71</td>
</tr>
<tr>
<td>cmlTagTableHeader</td>
<td>72</td>
</tr>
<tr>
<td>Hyperlinks</td>
<td>74</td>
</tr>
<tr>
<td>cmlTagHyperlink</td>
<td>74</td>
</tr>
<tr>
<td>cmlTagAnchor</td>
<td>78</td>
</tr>
<tr>
<td>Graphical Elements</td>
<td>78</td>
</tr>
<tr>
<td>cmlTagImage</td>
<td>78</td>
</tr>
<tr>
<td>cmlTagHorizontalRule</td>
<td>80</td>
</tr>
<tr>
<td>Other Elements</td>
<td>82</td>
</tr>
<tr>
<td>cmlTagClear</td>
<td>82</td>
</tr>
<tr>
<td>cmlTagCMLEnd</td>
<td>83</td>
</tr>
</tbody>
</table>
About This Document

This document contains information about the Palm Query Application (PQA) File Format. Information about Palm Database file (PDBs) formats and Palm Resource (PRCs) file formats will be added to this document within a month after this release and posted to the Palm website:

http://www.palm.com/devzone/docs.html
PQA Database Format

Overview

A Palm Query Application (PQA) is a Palm Database (PDB) containing world-wide web content. On the Palm device all PQAs are associated through the Launcher with the Clipper web-clipping viewer. When a user opens a PQA file for viewing, the Launcher starts Clipper, which in turn displays the contents of the selected PQA.

A PQA database may contain one or more PDB records, which in turn contain the actual web content. Clipper displays each record as a “page” in the sense that a traditional web browser displays an HTML file. Records contain either hypertext markup language (HTML) encoded into simplified PQA format, or image content encoded as Palm bitmap data. The data is also compressed. The content in a record may contain links to other records in the same PQA, to other PQAs or applications, and to HTML pages on a remote server.

This chapter describes the format and contents of a PQA.

The version described herein includes DatabaseHdrType version 1 and PQA header version 3.

NOTE: All structure elements in all headers are byte-packed in network (big-endian) order.

The offsets listed in the structure diagrams are in hexadecimal.
PQA Database Format

The format of a PQA database is shown in Figure 1.1. Note that this is a normal Palm database with certain constraints unique to a PQA, as described throughout this chapter.

A PQA contains the following parts:

- A standard PDB header (DatabaseHdrType). See “PDB Header for a PQA” on page 3.
- A single list (RecordListType) of records (RecordEntryType) that correspond to individual web content records. See “Record List in the PDB Header” on page 7.
- Two NULL bytes separating the end of the record entry list from the start of the appInfo block. This is an artifact of pre-existing PDB support in the Palm OS®, and is required by the HotSync® mechanism.
- An appInfo block, containing a PQA header. See “PQA appInfo Block” on page 8.
• One or more individual resource blocks (records) containing displayable web content. Each record contains a header (PqfWebDocRecordType) and web content (HTML data encoded into the PQA format, or graphic data). See “Web Content Record” on page 11.

PDB Header for a PQA

The PDB header in a PQA is a standard DatabaseHdrType structure. The format is shown in Figure 1.2.
A Palm OS application should use Palm OS functions for extracting data from particular records in the database and should have little need for details concerning the PDB structure. However, there are four elements in the PDB header that contain PQA specific data: name, attributes, type, and creator.
**Field Descriptions**

**name**
A string containing up to 31 bytes of the name and extension (filename.ext format) of the PQA file, which is also the name of the database on the Palm device. (This file is the one saved to disk by the Query Application Builder application on the content development machine.) This is the file name used to back up the PDB during the HotSync process to a desktop computer. The name string preserves the case of the original file name. This string is null-terminated; there are 31 characters available for the actual name.

**attributes**
A Word of flags; standard PDB attributes. For all PQAs:

- **attributes = dmHdrAttrBackup | dmHdrAttrLaunchableData**

The `dmHdrAttrBackup (0x0008)` attribute sets the backup bit. The `dmHdrAttrLaunchableData (0x0200)` attribute designates this PDB as one that can be launched by an application with the same creator, which is how PQAs are associated with the Clipper application (see creator below).

**version**
A Word; the version of the database layout. Currently 1 for the layout described here.

**creationDate**
A DWord; the creation date of the database. The time when the PQA was written to disk on the content development machine.

**modificationDate**
A DWord; the time of the last modification of the PQA. Initially set equal to the value of `creationDate`.

**lastBackupDate**
A DWord; the time the PQA was last backed up. Initially set to the value 2.

**modificationNumber**
A DWord; the modification number of the database. Initially set to 0.
PQA Database Format
PDB Header for a PQA

appInfoID
A LocalID (DWord); the offset from the beginning of the PDB header data to the start of the application-specific appInfo block, identifying the location of the PQA header. For a given PQA, this offset is equal to:
\[ \text{sizeof(DatabaseHdrType)} - 2 + \text{numRecords} \times \text{sizeof(RecordEntryType)} \]
Subtract two to allow for the two dummy bytes in the record entry list, recordList (see the following section, Record List in the PDB Header).

sortInfoID
A LocalID (DWord); offset from the beginning of the PDB header data to the start of an application-specific sortInfo block (PDB sorting information). Unused in PQAs; always 0.

type
A DWord; the PDB type identifier. Always 'pqa' (0x70716120).

creator
A DWord; the PDB creator identifier. Always 'clpr' (0x636C7072).
This, together with the dmHdrAttrLaunchableData attribute, identifies PQAs as databases to be launched by the Clipper application.

uniqueIDSeed
A DWord; normally used to generate unique identifiers on the Palm device. Unused in PQAs; always 0.

recordList
A RecordListType; a single record list. For details, see the following section, Record List in the PDB Header.
Normally, a PDB (not a PQA) recordList field may contain more than one list (RecordListType) of record entries. A PQA PDB header always contains a single list.

a. For creationDate, modificationDate, and lastBackupDate, the value is as given by the standard C run-time routine time() added to a constant offset (decimal 2082844800) to produce a Palm-device standard time value. The value is the number of seconds elapsed since midnight (00:00:00), January 1, 1970, coordinated universal time, according to the system clock. A Palm device standard time value is the number of seconds elapsed since 1/1/1904.
Record List in the PDB Header

A PDB header in a PQA file contains a standard record list, shown in Figure 1.3.

There is always a single record list in a PQA PDB header. The list consists of a header (a RecordListType) followed by 0 or more record entries (of type RecordEntryType).

Figure 1.3  PQA Record List

The fields in the RecordListType header are described here:

**Field Descriptions**

- **nextRecordListID**  A LocalID (DWord); the local offset of the next list. In the PDB header of a PQA, always 0 (i.e. a PQA PDB header always contains only a single record entry list).

- **numRecords**  A Word; the number of record entries in this list. Also, by definition, the number of web record resources contained in the PQA.

- **unused bytes**  Two pad bytes set to 0; not used.
Each record list entry (RecordEntryType) in the PDB header identifies a single web content record in the PQA. Here are the fields in each record list entry:

**Field Descriptions**

- **localChunkID**: A LocalID (DWord); the offset from the top of the PDB to the start of the web content record data for this entry. This is the offset to the start of the web content records's header (PqfWebDocRecordType).

- **attributes**: One byte; always 0 in a PQA.

- **uniqueID**: Three bytes; always 0 in a PQA.

**PQA appInfo Block**

Each PQA’s PDB header (described previously) is followed by a block of application-specific data.

The format is shown in Figure 1.4. The field names followed by asterisks (*) identify variable-length fields of bytes, padded, if necessary, to a word boundary.
Figure 1.4 PQA applInfo Block

Field Descriptions

- **signature**: A DWord; must be 'lnch' (0x6C6E6368).
- **hdrVersion**: A Word; the version of this PQA header. Currently 3.
- **encVersion**: A Word; for PQAs that contain HTML data encoded into the PQA format, the version of the encoding. All web content records within a given PQA are taken to have the same encoding version.
- **verStrWords**: A Word; the length of the verStr (version string) that follows, in 16-bit words, including any pad byte at the end.
**verStr**
A sequence of \((\text{verStrWords} \times 2)\) Bytes; a null-terminated version string that Clipper shows. This is the version string for the PQA itself (i.e. the version of this web content).
If the value of \text{verStrWords} is zero, the \text{verStr} field will contain no bytes.
The end of this sequence of bytes must be word-aligned. If the size of the data (including the string’s null terminator) is an odd number of bytes, the data must be followed by a null pad byte.

**pqaTitleWords**
A Word; the length of the \text{pqaTitleStr} (PQA Launcher-visible title string) that follows, in 16-bit words, including any pad byte at the end.

**pqaTitleStr**
A sequence of \((\text{pqaTitleWords} \times 2)\) Bytes; a null-terminated title string that the Launcher shows for this PQA’s icon. This is the title string for the PQA itself; not the title string included in the original HTML index file’s source for the web page’s title, shown by Clipper.
If the value of \text{pqaTitleWords} is zero, the \text{pqaTitleStr} field will contain no bytes.
The end of this sequence of bytes must be word-aligned. If the size of the data (including the string’s null terminator) is an odd number of bytes, the data must be followed by a null pad byte.

**iconWords**
A Word; the length of the bitmap data that follows, in 16-bit words, including any pad byte at the end.

**icon**
A sequence of \((\text{iconWords} \times 2)\) Bytes; a Palm bitmap (\text{BitmapType} and associated bitmap data). This is the large icon that appears on the device for this PQA.
If the value of \text{iconWords} is zero, the icon field will contain no bytes.
The end of this sequence of bytes must be word-aligned. If the size of the data is an odd number of bytes, the data must be followed by a null pad byte.
Web Content Record

Following the appInfo block in a PQA is a sequence of web content records; one for each record list entry in the PDB header record list. Each web content record begins on a word boundary, and contains a header (PqfWebDocRecordType) followed by the content’s original URL and the content itself (that is, HTML data encoded into the PQA format, or graphic data).

The layout of a web content record is shown in Figure 1.5.
Figure 1.5  Web Content Record
Field Descriptions

urlOffset
A DWord; the offset, in number of bytes, from the top of this PqfWebDocRecordType header to the start of the URL for this web content resource. For this version of the web content record, this field’s value is always 0x14. This seemingly unnecessary field is included in this version of the record format for historical reasons.

urlLength
A Word; the length of the URL string, in bytes. This is simply the size of the URL string itself and counts neither a null terminator nor any pad byte following the string data.

dataOffset
A DWord; the offset, in number of bytes, from the top of this PqfWebDocRecordType to the start of the data for this web content resource.

dataLength
A Word; the length of the data, in bytes.

contentType
A Byte; a code for the type of content, defined in CMLConst.h and described in Chapter 2, “Content And Compression Types.” The contentType indicates the type of resource encoded in this record, e.g. HTML, text, JPEG, or GIF. The content type is determined either from a MIME string passed from a server or by the filename extension of the original resource. Valid content types include:
cmlContentTypeTextPlain
cmlContentTypeTextHTML
cmlContentTypeImageGIF
cmlContentTypeImageJPEG
cmlContentTypeTextCml
cmlContentTypeImagePalmOS

Version 1 of the Palm Query Application Builder application encodes any valid image resource (i.e. GIF or JPEG) into cmlContentTypeImagePalmOS, and all other resources as cmlContentTypeTextCml. Other content types are provided for future use.
### PQA Database Format

#### Web Content Record

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compressionType</td>
<td>A Byte indicating the type of compression of the content, defined in CMLConst.h. Valid types include: cmlCompressionTypeNone, cmlCompressionTypeBitPacked, cmlCompressionTypeLZ77. Type cmlCompressionTypeNone is loosely termed “raw” and is described in “Unpacked Notation” on page 37. Type cmlCompressionTypeBitPacked is the standard PQA compression type, described in “Bit Packed Compression” on page 24. Type cmlCompressionTypeLZ77 is reserved for future use.</td>
</tr>
<tr>
<td>uncompressedDataSize</td>
<td>A DWord; the uncompressed size, in bytes, of the web content. This field contains the size of the resource data in “raw” (cmlCompressionTypeNone) form, before compression is applied to produce data using cmlCompressionTypeBitPacked compression. Note that in test code, in a record with compressionType set to cmlCompressionTypeNone, the value of the field uncompressedDataSize equals the value of the field dataLength. If the web content is an image, this field contains the size of the Palm OS bitmap data before compression.</td>
</tr>
<tr>
<td>flags</td>
<td>A Byte; currently unused. Should be zero.</td>
</tr>
<tr>
<td>reserved</td>
<td>A Byte; currently unused. Should be zero.</td>
</tr>
</tbody>
</table>
URL string

The URL string follows the end of the PqfWebDocRecordType structure. The string contains the filename of the individual resource as it existed on the development system when the PQA was built (for example, “palm.htm”). The URL string may be followed by a zero pad byte (not a null-terminator), if necessary, to align the string data on a word boundary.

Web document data

Document data begins on a word boundary following the end of the URL string. Version 1 of the Palm Query Application Builder application encodes any valid image resource (i.e. GIF or JPEG) into cmlContentTypeImagePalmOS, and all other resources as cmlContentTypeTextCml. A cmlContentTypeImagePalmOS resource is a standard Palm bitmap, compressed using standard Palm bitmap image compression. A cmlContentTypeTextCml resource is text or HTML data encoded in the PQA format, described in Chapter 3, “PQA Encoding Format.” The Palm Query Application Builder application builds cmlContentTypeTextCml resources that are compressed using cmlCompressionTypeBitPacked compression.
PQA Database Format
Web Content Record
The Palm web clipping application (Clipper) displays web content on a Palm VII™ device. This content arrives on the device either within a Palm query application (PQA) database or as part of a transmission from the Palm Web Clipping Proxy server.

The container identifies the type of content it contains. The valid type identifiers are:

- cmlContentTypeTextPlain 0
- cmlContentTypeTextHTML 1
- cmlContentTypeImageGIF 2
- cmlContentTypeImageJPEG 3
- cmlContentTypeTextCml 4
- cmlContentTypeImagePalmOS 5

The current version of the Clipper web clipping application processes only content identified with either content type cmlContentTypeTextCml or cmlContentTypeImagePalmOS.

A PQA or proxy server response stream also identifies the compression type applied to the content. The valid compression type identifiers are:

- cmlCompressionTypeNone 0
- cmlCompressionTypeBitPacked 1
- cmlCompressionTypeLZ77 2

The current version of the Clipper web clipping application processes only content identified with either compression type cmlCompressionTypeNone or cmlCompressionTypeBitPacked.
The Palm PQA data format encoder converts resources with MIME type text/html to content with type `cmlContentTypeTextCml`. The format of the data is specified in Chapter 3, “PQA Encoding Format.”

**Plain Text**

The data format encoder interprets resources with MIME type text/plain, as well as any content that is not identified as text/html, image/gif, or image/jpeg data, as “plain text” content. The encoder processes the source content and produces plain text consisting only of characters that fall within the defined ANSI text character set (0x20 through 0x7e, 0x82 through 0x8c, 0x91 through 0x9e, and 0xa1 through 0xff) together with ASCII tab (0x09), linebreak (0x0a), and carriage return (0x0d) codes. The encoder identifies this content as type `cmlContentTypeTextCml` since that is the only non-image content type that Clipper handles.

**cmlContentTypeImagePalmOS**

Content type `cmlContentTypeImagePalmOS` is standard Palm OS® bitmap image data, which may be compressed according to the Palm OS bitmap standard. The current version of the Palm PQA data format encoder, used by both the Palm Query Application Builder application and by the Palm Web Clipping Proxy server, converts resources with MIME content types image/gif and image/jpeg into compressed Palm OS bitmaps.

See `BitmapType` and `BitmapFlagsType` in the header file `WindowNew.h` for information on the Palm OS bitmap format.

**cmlCompressionTypeNone and cmlCompressionTypeBitPacked**

Type `cmlCompressionTypeNone` is an intermediate form; the Query Application Builder and Palm Web Clipping Proxy server always generate `cmlCompressionTypeBitPacked` data.
Content And Compression Types

cmlCompressionTypeNone and cmlCompressionTypeBitPacked

For more information, see the sections “Bit Packed Compression” on page 24 and “Unpacked Notation” on page 37.
Content And Compression Types

cmlCompressionTypeNone and cmlCompressionTypeBitPacked
PQA Encoding Format

This chapter describes the PQA data encoding format.

The Clipper web clipping display application on the Palm VII device, the Palm Query Application Builder application, and the Palm Web Clipping Proxy servers all work with a data format that is a binary translation of HTML source data, known as the PQA data format. Furthermore, the Query Application Builder application and proxy servers produce data that is compressed with a Palm-proprietary “bit-packed” scheme.

This chapter describes the PQA data encoding format and the associated bit-packed compression scheme, and includes the specifications for both. Note that when it’s necessary to discuss unpacked data, we’ll refer to the data as raw, unpacked or uncompressed.

This document describes version 1 of the PQA encoding and compressing scheme. Future versions may include additional features and support more content types.

Overview

PQA data is a stream of text and image data with embedded formatting tags. PQA data is generated from HTML data; PQA tags embedded in the data correspond to HTML tags. For example, some common HTML tags (BR, P, DIV) are mapped to single linebreak characters; other PQA tags and their parameters are embedded as binary data rather than ASCII characters (see the section “Unpacked Notation” on page 37).

Given some HTML input, a PQA data format encoder transforms HTML tags to their PQA representations, ignoring unsupported HTML tags, and converts images to Palm OS® bitmaps to be
embedded in the PQA output. (Note that the current version of the PQA data format encoder used by the Palm Query Application Builder and the Palm Web Clipping Proxy server accepts and converts only GIF and JFIF images.) The result is uncompressed PQA format data.

After transforming the HTML source to a PQA representation, the encoder may compress the data using the bit-packed compression scheme.

Here is an example translation from HTML to bit-packed PQA format. Given this original HTML content:

```html
<html>
<head>
  <title>Example</title>
</head>
<body>
  Body text
</body>
</html>
```

A byte stream containing unpacked PQA data appears as follows (in hexadecimal):

```
45 78 61 6D 70 6C 65 00 42 6F 64 79 20 74 65 78 74 01 71
```

A bit stream containing the equivalent bit-packed data appears as follows:

```
0001001000101111100111010100101010101000010101000000000000100100
00101010001001111110010111001010101010110110010000101111001
```

Or, in hexadecimal, it appears like this:

```
12 2F 4D 2A C5 40 12 15 13 E2 E5 5D C8 5C 40
```
Encoding the HTML as bit-packed PQA data results in a compact representation relative to the size of the original. (Note that the hexadecimal representation above includes zero bits not part of the PQA bit stream, which actually ends with the last “on” (1) digit in the byte with the value 40h.)

Here is a break-out of the data elements for comparison:

| 00010 | <single character escape> |
| 01000101 E |
| 11101 x |
| 00110 a |
| 10010 m |
| 10101 p |
| 10001 l |
| 01010 e |
| 00000 <title string textz null terminator> |
| 00010 <single character escape> |
| 01000010 B |
| 10100 o |
| 01001 d |
| 11110 y |
| 00101 <space> |
| 11001 t |
| 01010 e |
| 11101 x |
| 11001 t |
| 00001 <start of tag> |
| 01110001 cmlTagCMLEnd |

This specification describes how HTML elements are encoded and compressed into PQA format to produce a bit stream like the one shown here.

**Differences From HTML**

The major emphasis of the PQA format is that it is optimized for size, in keeping with the overall goal for the Palm VII device of minimizing the number of bytes transmitted over the air. The PQA
format consists of binary data. Readability and flexibility are compromised for compactness.

One major design difference between HTML and PQA format is that PQA format is not designed as a content creation language. The PQA format is not intended to be used to actively mark-up web content. It is merely a temporary format used to represent content as it is being transferred between a server and a client. As such, it is always algorithmically generated from HTML source, a process similar to object code being generated from a compilation of source code.

Another important difference between PQA format and HTML is that white space and line breaks in the PQA format text are significant. That is, the equivalent of the HTML line break tag (<BR>) is not required in PQA format since line breaks are embedded directly into the text as linebreak characters.

Lastly, unlike HTML, the PQA data format specifies no identification scheme of any kind; successful data transfer and handling depends entirely upon context. There is no header or magic number at the start of a stream of PQA data, unless such identification is part of some enclosing transport mechanism for the data. For example, PQA data is expected in a response from the Palm Web Clipping Proxy server and within a Palm Query Application resource, and is identified by the appropriate headers in each case.

For details on the HTML tags and attributes that are supported in the Palm system, refer to Appendix A of the Web Clipping Developer’s Guide.

**Bit Packed Compression**

In its raw form, unpacked PQA data (cmCompressionTypeNone) is an encoded form of HTML, smaller in size than the original content. Of course, a stream of unpacked encoded data can be compressed further. Although unpacked PQA data could be compressed using any standard method, for historical reasons the current versions of the Palm Query Application Builder application, the Palm Web Clipping Proxy servers, and the Clipper web clipping display application, all
work with PQA data that has been compressed using a proprietary compression scheme called “bit-packed” (cmlCompressionTypeBitPacked).

Palm software uses the bit-packed scheme in the hopes of reducing the size of the web content transmitted over the air. This is in keeping with one of the primary goals of the Palm VII device: to minimize the number of bytes transmitted, and thus to minimize data transmission charges.

The fundamental idea behind bit-packed compression is simply to map single- or multiple-byte data elements in an unpacked PQA data stream to data elements represented by fewer bits in a bit stream.

A bit-packed PQA data stream is by default a 5-bit character text stream. That is, until a special character (see below) appears in the stream, each sequence of 5 bits is assumed to represent a single text character. Table 3.1 lists the possible 5-bit characters.

<table>
<thead>
<tr>
<th>Value</th>
<th>Special</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Yes</td>
<td>Yes</td>
<td>EndTag character (cmlCharEnd). Used to end a TextZ type and certain tags.</td>
</tr>
<tr>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>StartTag character (cmlCharStart), followed by an 8-bit Tag ID</td>
</tr>
<tr>
<td>2</td>
<td>Yes</td>
<td>No</td>
<td>Single character escape (cmlCharEsc), followed by a single ASCII character</td>
</tr>
<tr>
<td>3</td>
<td>No</td>
<td>No</td>
<td>ASCII Formfeed (0x0c), (cmlCharFormFeed)</td>
</tr>
<tr>
<td>4</td>
<td>No</td>
<td>No</td>
<td>ASCII Carriage return (0x0d), (cmlCharLineBreak)</td>
</tr>
<tr>
<td>5</td>
<td>No</td>
<td>No</td>
<td>ASCII Space (0x20), (cmlCharSpace)</td>
</tr>
<tr>
<td>6-31</td>
<td>No</td>
<td>No</td>
<td>ASCII lowercase letters: a through z (0x61 through 0x7a)</td>
</tr>
</tbody>
</table>

The table columns have the following meanings:
Value is the 5-bit numeric value.

Special indicates whether or not the value is an encoding escape or text. As described under “cmlTag8BitEncoding” on page 46, an encoder may produce sections of output within which text is encoded using 8 bits per character instead of 5 bits per character. Even in these sections with larger number of bits per character, the decimal character values 0 through 2 always have special meaning.

Reset indicates whether or not a decoder that is currently processing a cmlTag8BitEncoding of 8-bit text characters should reset to 5-bit mode when the decoder encounters a “reset” character. (See “Tag Definitions” on page 41 for details of the tag encodings.)

A bit-packing decompressor operates essentially in three modes:

- 5-bit character mode, in which each group of 5 bits of input is interpreted as one of the bit-packed encoding characters.
- Single-character escape mode, in which the next 8 bits of input is taken as a single character.
- Tag mode, in which the bits of input are interpreted according to the tag encoding definitions given in this specification (see the section “Tag Definitions” on page 41.)

Bit-packing provides a benefit when applied to input consisting of lowercase ASCII text characters and HTML tags and attribute values (including image data). In the case of lowercase characters, each 8-bit character is translated into a 5-bit equivalent. In the case of HTML tags, numeric tag attribute values may be compressed with variable-length integer encoding. In the case of image data, the standard Palm OS bitmap image compression scheme uses bit-packing.

Conversely, bit-packing results in expansion when applied to non-lowercase characters (e.g. uppercase and numeric characters). In this case, any reduction of the data size of an HTML stream encoded as a bit-packed PQA data stream is due to the PQA encoding itself.

Note that the 5-bit compressor used by the Palm Query Application Builder and the Palm Web Clipping Proxy servers takes only ASCII text or uncompressed PQA data as input. That compressor does not directly interpret HTML tags and end-tags in the input, and also
generates bit-packed plain text only from plain ASCII text data or as part of PQA data.

Following is an example of how a simple section of text would be represented in PQA format. The text:

```
abc d
ef
```

would be represented as:

```
Bit[5] char = 6  // 'a'
Bit[5] char = 7  // 'b'
Bit[5] char = 8  // 'c'
Bit[5] char = 5  // '
Bit[5] char = 9  // 'd'
Bit[5] char = 4  // line break
Bit[5] char = 10 // 'e'
Bit[5] char = 11 // 'f'
```

which, as a binary bit stream is:

```
00110 00111 01000 00101 01001 00100 01010 01011
```

When the text encoding mode is 5 bit characters, the single character escape (2), is always followed by an 8 bit ASCII character. This single character escape then can be used to represent characters which are not present in the 5-bit alphabet. For example, the text:

```
a Cow
```

would be represented in PQA format as the following sequence:

```
Bit[5] char = 6  // 'a'
Bit[5] char = 5  // '
Bit[5] char = 2  // single character escape
Bit[8] char = 67 // 'C'
Bit[5] char = 20 // 'o'
Bit[5] char = 28 // 'w'
```

where the 67 is the 8 bit sequence 01000011 which represents the ASCII value for 'C' (67 decimal, 0x43 hexadecimal), and all other characters are 5 bits long.
Multiple sequences of non-lower case alpha or international characters can also be included in the stream by including the appropriate text encoding tag in the stream followed by the 8 or 16 bit (unicode) character text string. Tags are described in the next sections.

**Bit Packed Compression Encoding**

The bit-packed compression scheme uses four fundamental encodings for each of ASCII text, HTML tags, numeric parameter values of the tags, and images.

These four encodings are described in the following sections. ASCII, tag, and numeric parameter value encodings are defined in the rest of this specification.

**ASCII Text Encoding**

Lowercase ASCII text characters, the space character and the linebreak character are mapped to corresponding 5-bit codes. All other ASCII text characters are encoded either by a 5-bit single-character escape code or within a tagged run of 8-bit ASCII characters.

**Tag Encoding**

All HTML tags are encoded as:

- 5-bit “start tag” code
- 8-bit tag identifier

If the tag includes attributes, the encoding includes:

- Encoded tag parameters (using numeric parameter value and ASCII text encodings)

If the tag encloses associated tag data, for example, a hyperlink tag enclosing a link or an image tag specifying an image URL, the encoding includes:

- Encoded tag data (using ASCII text encoding and image compression)

If the tag requires an end tag, for example, a hyperlink tag's </A>, the encoding includes:
• 5-bit “end tag” code

Numeric Parameter Value Encoding

Numeric HTML parameter values may be compressed by encoding the numeric values as binary numbers. Further, the binary representations may be further compressed using variable-length integer representations, defined under “Compact Data Structure Types” on page 30.

Image Compression

The encoder must convert all original source image data to Palm OS bitmap data. A bit-packing compressor compresses all Palm OS images in the data with standard Palm OS bitmap image compression, which is somewhat akin to PICT image compression on the Mac OS.

Compact Data Structure Notation

Throughout the rest of this document, PQA data is represented using a notation similar to that used in the C language for representing data structures. We’ll call this notation Compact Data Structure Notation (CDSN).

This notation describes PQA data elements with compression type cmlCompressionTypeBitPacked.

The general form is:

\[
\text{<data type>} \text{ <identifier> = <legal value>}\]

For example, the notation for a three bit value:

\[
\text{Bit[3] aValue = 7}\]

Note that \text{<legal value>} may be an identifier the value of which is a legal value. Also, note that the values of \text{Bit[5]} arrays are typically denoted by the numeric values of characters defined in bit-packed encoding, and given as the code for that character (for example, 6 for 'a', 0 for the end tag code, etc.).

Here’s a longer example:

\[
\begin{align*}
\text{Bit enabled} &= 1 \\
\text{Bit[3] type} &= \text{typeRound}
\end{align*}
\]
Int16 length = 0x1234

The above structure represents the following sequence of 20 bits:

```
1 010 0001001000110100
```

The first field, enabled, is a 1 bit field that has the value 1. The second field, type, is a 3 bit field that has the value typeRound which is a constant defined to be 2. The third field, length, is a 16-bit integer with the value 0x1234.

Fields in CDSN are never padded to fall on word or byte boundaries. That is, each field starts off on the next free bit after the previous field. All multi-bit values are stored most-significant-bit first.

**Compact Data Structure Types**

A number of primitive data types are used in Compact Data Structure Notation. The basic ones are:

- **Bit**: a single bit
- **UInt8, Int8**: 8 bit unsigned and signed integers
- **UInt16, Int16**: 16 bit unsigned and signed integers
- **UInt32, Int32**: 32 bit unsigned and signed integers

Other important types include the variable length integer types: **UIntV** and **IntV**. These can be anywhere from 1 to 36 bits in length, depending on their value. The actual length can be determined by looking at the first 1 to 4 bits.

Using the **UIntV**, an integer of value 0 can be represented with just 1 bit, values 1 through 7 would require 5 bits, values 8 through 63 would require 9 bits, etc.

The types **UIntV, IntV, UInt16V, Int16V, UInt8V, and Int8V** are defined in the following sections.

**UIntV**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The value 0</td>
</tr>
<tr>
<td>10</td>
<td>Bit[3]</td>
</tr>
<tr>
<td></td>
<td>The values 0 through 7 (0x07)</td>
</tr>
</tbody>
</table>
110  Bit [6]  The values 0 through 63 (0x3F)
1110  Bit [16]  The values 0 through 65535 (0xFFFF)
1111  Bit [32]  The values 0 through 4,294,967,295 (0xFFFFFFFF)

**IntV**

0  The value 0
10  Bit [3]  The values -4 through 3
110  Bit [6]  The values -32 through 31
1110  Bit [16]  The values -32768 through 32767
1111  Bit [32]  The values -2,147,483,648 through 2,147,483,647

**Uint16V**

0  The value 0
10  Bit [3]  The values 0 through 7 (0x07)
110  Bit [6]  The values 0 through 63 (0x3F)
1110  Bit [16]  The values 0 through 65535 (0xFFFF)

**Int16V**

0  The value 0
10  Bit [3]  The values -4 through 3
110  Bit [6]  The values -32 through 31
1110  Bit [16]  The values -32768 through 32767

**Uint8V**

0  The value 0
10  Bit [3]  The values 0 through 7 (0x07)
PQA Encoding Format

PQA Tags

The tag start character (1) is included in a data stream to indicate the presence of a PQA tag. It is always followed by an 8-bit Tag ID structure, and optionally followed by other variable length bit fields, depending on the specific tag. The 8-bit Tag ID structure can have a value of 0 through 255 (0 through 0xFF).

Different tags have different functions. Some tags are always followed by other variable length bit fields which specify parameters for that particular tag function. Other tags have no parameters at all. In any case, because the tag start character is a reset character, the text encoding mode is always set back to 5-bit characters whenever a tag is encountered (unless the tag specifically changes the text encoding mode).

This specification describes PQA data elements with their bit-packed representations.

For example, the tag cmlTagTextBold is used to turn on bold formatting. It has no parameters. The following text:

```
a  cow
```

would be represented in PQA data format as:

```
Bit[5] char = 6 // 'a'
Bit[5] char = 5 // '

Bit[5] char = 1 // tag escape character
Bit[8] tagID = cmlTagTextBold // bold constant
Bit[5] char = 8 // 'c'
```
An example of a tag which has parameters is the cmlTagTextSize tag. This tag is always followed by a UIntV specifying the actual text size to use. For example, the following text:

   a dog

would be represented in PQA format as:

   Bit[5] char = 6 // 'a'
   Bit[5] char = 5 // '
   Bit[5] char = 1 // tag escape character
   Bit[8] tagID = cmlTagTextSize // size constant
   Bit[3] size = 4
   Bit[5] char = 9 // 'd'
   Bit[5] char = 20 // 'o'
   Bit[5] char = 12 // 'g'

Text Encoding Tags

Some PQA tags are used to include strings of text that cannot be encoded as 5-bit characters. Conceptually, text encoding tags are merely tags that have a variable number of parameters following them, where each “parameter” is another character in the text stream. The sequence of “parameters” ends as soon as a reset character is encountered (the cmlCharEnd or cmlCharStart character).

For example, the cmlTag8BitEncoding tag indicates a string of 8 bit characters follows. The string of 8 bit characters is assumed to continue in the stream until a reset character is encountered. However, because the stream is now built up of 8 bit characters, all special characters (which includes the reset characters and single character escape) are also now 8 bits long. For example, the cmlCharEnd character becomes the 8 bit sequence 0b00000000 and the cmlCharStart character becomes the 8 bit sequence 0b00000001.

In all cases of alternate text encodings, as soon as a reset character (0 or 1 decimal) is encountered in the stream, the text mode is switched back to 5 bit characters.
The following is an example of how the `cmlTag8BitEncoding` tag is used. The text:

```
a BIG dog
```

would be represented in PQA format as:

```plaintext
Bit[5] char = 6 // 'a'
Bit[5] char = 5 // 'I'
Bit[5] char = 1 // tag escape character
Bit[8] tagID = cmlTag8BitEncoding
Bit[8] char = 'B' // 'B'
Bit[8] char = 'I' // 'I'
Bit[8] char = 'G' // 'G'
Bit[8] char = 0 // cmlCharEnd, switches text
               // encoding back to 5-bit mode
Bit[5] char = 9 // 'd'
Bit[5] char = 20 // 'o'
Bit[5] char = 12 // 'g'
```

An important thing to note is the interaction of alternate text encoding sections with the `cmlCharEnd` character. Besides being used as a way to reset the text encoding mode, the `cmlCharEnd` character is sometimes used to separate two elements or to indicate the end of a block level element.

For example, when a list needs to be represented in PQA format, the list items are separated from each other by the `cmlCharEnd` character. In these instances, if a list item was represented using 8-bit encoded text, there would be two `cmlCharEnd` characters in a row in the stream. The first `cmlCharEnd` character, needed to end the 8-bit encoded text, would be 8 bits long. Then, to indicate the actual start of another list item, a 5-bit `cmlCharEnd` character would be placed in the stream.
NOTE: Since there is a fixed amount of overhead associated with adding a `cmlCharStart`, `cmlTag8BitEncoding`, and `cmlCharEnd` to a run of characters, the Query Application Builder application and the Palm Web Clipping Proxy servers do not always encode consecutive runs of uppercase characters as eight-bit encodings. Runs of uppercase characters of length four or less are encoded as sequences of single character escapes. Runs of length five or more are encoded as 8-bit encodings.

The Tag Data Type

Because the sequence of the tag escape character followed by a Tag ID structure is used so often in the documentation, it is given its own data type. It is defined as:

```c
Tag tagID:
    Char cmlCharStart = 1
    Bit[8] tagID
```

Text & TextZ Types

Another common data type is the `Text` data type. This type is used to represent a string of characters. This type is a powerful data type because it hides the complexity of escaping special characters and the actual number of bits required to represent each character.

For example, here is the sequence used above:

```c
Bit[5] char = 6 // 'a'
Bit[5] char = 5 // '
Bit[5] char = 1 // tag escape character
Bit[8] tagID = cmlTag8BitEncoding
Bit[8] char = 'B' // 'B'
Bit[8] char = 'I' // 'I'
Bit[8] char = 'G' // 'G'
Bit[8] char = 0 // cmlCharEnd character
Bit[5] char = 9 // 'd'
Bit[5] char = 20 // 'o'
Bit[5] char = 12 // 'g'
```
This can be represented using the Text data type as:

    Text string = "a BIG dog"

Notice, that the Text data type hides the complexities of escaping non-lower case alpha characters as well as the cmlCharEnd character used to switch the mode back from 8-bit to 5-bit ASCII.

The combination of the Tag and Text types makes representing combinations of formatting and text sequences much easier. For example, the sequence used above that showed how bold text would be represented was:

    Bit[5] char = 6 // 'a'
    Bit[5] char = 5 // '
    Bit[5] char = 1 // tag escape character
    Bit[8] tagID = cmlTagTextBold // bold constant
    Bit[5] char = 8 // 'c'
    Bit[5] char = 20 // 'o'
    Bit[5] char = 28 // 'w'

Using the Tag and Text types, this sequence can be represented as:

    Text string = "a 
    Tag tag = cmlTagTextBold
    Text string = "cow"

**TextZ Type**

Another convenient type is the TextZ type. This is basically a Text type with a terminating cmlCharEnd character. This type is most often used in tag parameter lists. It can be defined simply as:

    TextZ text:
    Text text
    Char end = cmlCharEnd

As an example, the format of the anchor tag is defined as:

    Tag tag = cmlTagAnchor
    TextZ name

Where the name parameter is a string holding the local anchor name. This string is delimited from any text that might follow the tag by the cmlCharEnd character at the end of it. If a variable is
defined as a TextZ type, it must have a cmlCharEnd character at the end of it.

Unpacked Notation

Originally, the PQA format was envisioned as a tag-encoding method with one representation: what is currently the cmlCompressionTypeBitPacked compressed form. Later, it became apparent that it would be advantageous to define a byte-aligned uncompressed, or unpacked, representation for debugging purposes. This unpacked form became the cmlCompressionTypeNone form.

Unpacked PQA format then was defined to consist of only the tag encoding. PQA data is thus representable in two forms: unpacked and bit-packed compressed. In unpacked form, HTML tags are encoded as PQA tags, including start and end tag characters in byte form. In bit-packed compressed form, text characters (ASCII text, start and end tag characters), tag attribute values, and image data are encoded according to the bit-packed compression scheme.

The encoding module used by the Palm Query Application Builder application and by the Palm Web Clipping Proxy server encodes data in two passes. In the first pass, HTML is encoded as unpacked data (cmlCompressionTypeNone). In the second pass, a bit-packing compressor produces bit-packed (cmlCompressionTypeBitPacked) data.

If one is writing or debugging a PQA encoder, during debugging, you will probably find it useful to view and interpret the intermediate cmlCompressionTypeNone data. This is much easier to debug.

NOTE: This documentation and its notation denotes bit-packed compressed content. You must interpret definitions of bit-packed elements to produce the equivalent unpacked elements.

The following sections describe how to interpret the bit-packed notation in this document to identify data elements of cmlCompressionTypeNone.
Translation of Bit-Packed to Uncompressed Data

Equivalent content with no compression can be directly translated from the bit-packed element definitions.

Unpacked PQA data includes just two special characters, as shown in Table 3.2.

Table 3.2 Unpacked Encoding Characters

<table>
<thead>
<tr>
<th>Value</th>
<th>Special</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Yes</td>
<td>Yes</td>
<td>cmlCharEnd character. Used to end TextZ data and certain tags.</td>
</tr>
<tr>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>cmlCharStart character, followed by an 8 bit Tag ID.</td>
</tr>
</tbody>
</table>

The translation from bit stream data in `cmlCompressionTypeBitPacked` form to byte-oriented data in `cmlCompressionTypeNone` form is straightforward:

- All bit-packed data elements less than 8 bits in width are coded as one byte.
- All ASCII data is coded as 8-bit.
- All variable length `UIntV` and `IntV` types are encoded using four bytes (`DWord`).
- All variable length `Uint16V` and `Int16V` types are encoded using two bytes (`Word`).
- All variable length `Uint8V` and `Int8V` types are encoded using one byte.
- Palm bitmap image data is uncompressed, and no `uncompressedDataSize` value follows the header bytes, as it does in the compressed form of the bitmap.
- The single character escape and the tag `cmlTag8BitEncoding` are never used in a `cmlCompressionTypeNone` byte stream.

All other characters are encoded in their ASCII form.
Here are examples of possible bit-packed data elements and equivalent uncompressed translations:

<table>
<thead>
<tr>
<th>Bit-packed Data</th>
<th>Uncompressed Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit = 1</td>
<td>Byte = 0x01</td>
</tr>
<tr>
<td>Bit[3] = 1, 0, 1</td>
<td>Byte = 0x05</td>
</tr>
<tr>
<td>TextZ = &quot;foo&quot;</td>
<td>&quot;foo&quot;, NULL terminated ASCII string</td>
</tr>
<tr>
<td>Byte = 0xCD</td>
<td>Byte = 0xCD</td>
</tr>
<tr>
<td>IntV = -1</td>
<td>DWord = 0xFFFFFFFF (-1)</td>
</tr>
<tr>
<td>UIntV = 7</td>
<td>DWord = 0x00000007</td>
</tr>
<tr>
<td>Uint8V = 2</td>
<td>Byte = 0x02</td>
</tr>
</tbody>
</table>

Five-bit tags are treated in the following manner:

<table>
<thead>
<tr>
<th>Bit-packed Data</th>
<th>Uncompressed Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmlCharEnd (0)</td>
<td>Byte = 0x00</td>
</tr>
<tr>
<td>cmlCharStart (1)</td>
<td>Byte = 0x01</td>
</tr>
<tr>
<td>cmlCharEsc (2)</td>
<td>Unused</td>
</tr>
<tr>
<td>cmlCharFormFeed (3)</td>
<td>Byte = 0x0C</td>
</tr>
<tr>
<td>cmlCharLineBreak (4)</td>
<td>Byte = 0x0D</td>
</tr>
<tr>
<td>cmlCharSpace (5)</td>
<td>Byte = 0x20</td>
</tr>
<tr>
<td>cmlTag8BitEncoding</td>
<td>Unused</td>
</tr>
<tr>
<td>cmlCharA (6)...cmlCharZ (31)</td>
<td>Byte = 0x61 ... 0x7a</td>
</tr>
</tbody>
</table>

One can see that there is not a one-to-one mapping from elements of a bit-packed data stream to elements of an unpacked data stream. For example, bit-packed data includes single character escapes, 8-bit character runs and variable-length integers; data encoded without bit-packed compression does not include these escapes and number packings. In other words, the special escape characters and bit
encodings are part of the bit-packed compression scheme only, not part of the uncompressed encoding scheme.

**Example Translation**

Here is an example translation from bit-packed data to unpacked data. The encoding from the previous example is shown.

Given a bit stream containing this bit-packed data (shown here with a break-out of the data elements):

```plaintext
00010 <single character escape>
01000101 E
11101 x
00110 a
10010 m
10101 p
10001 l
01010 e
00000 <title string textz null terminator>
00010 <single character escape>
01000100 B
10100 o
01001 d
11110 y
00101 <space>
11001 t
01010 e
11101 x
11001 t
00001 <start of tag>
01110001 cmlTagCMLEnd
```

A byte stream containing the unpacked data (shown in hexadecimal) appears as follows:

```
45 78 61 6D 70 6C 65 00 42 6F 64 79 20 74 65 78 74 01 71
Example\0 Body\0 text\0 cmlEnd
```
Data Termination

A PQA format data stream is terminated with a `cmlTagCMLEnd` tag.

As an aside, note that a byte buffer containing the complete, terminated bit stream from the previous section, shown as hexadecimal, appears as follows:

```
12 2F 4D 2A C5 40 12 15 13 E2 E5 5D C8 5C 40
```

The hexadecimal above shows that the buffer includes extra zero bits at the end to pad the stream's content to a byte boundary. This padding is not part of the PQA format specification; the stream actually ends with the last 1 bit of the `cmlTagCMLEnd` value. A decoder would discard all bits following the `cmlTagCMLEnd`.

Tag Definitions

This section lists the various PQA tags available. Each tag is described in detail along with its parameters, if any. This section refers to tags by name, but in the actual implementation a predefined constant is associated with each tag.

Tags are either end tag delimited, meaning that their effects are ended by an appropriate `cmlCharEnd` character, or not, meaning that their effects persist until another tag of that type is encountered.

Background Attributes

**cmlTagBGColor**

| **Description** | Sets the background color. |
| **End Tag Delimited** | No |
| **Parameters** | **Byte red** | A value from 0 to 255 that indicates the amount of red in the color. |
Byte green  A value from 0 to 255 that indicates the amount of green in the color.

Byte blue   A value from 0 to 255 that indicates the amount of blue in the color.

Example  
Tag tag = cmlTagBGColor  
Byte red = 0xFF  
Byte green = 0x80  
Byte blue = 0x80

Text Attributes

cmlTagTextColor

Description  Sets the text color.

End Tag  No

Delimited

Parameters  Byte red  A value from 0 to 255 that indicates the amount of red in the color.

Byte green  A value from 0 to 255 that indicates the amount of green in the color.

Byte blue   A value from 0 to 255 that indicates the amount of blue in the color.

Example  
Tag tag = cmlTagTextColor  
Byte red = 0xFF  
Byte green = 0x80  
Byte blue = 0x80

Text "This text is reddish"


cmlTagLinkColor

Description  Sets the text color used to display unvisited, visited, and active links.
Tag Definitions

End Tag Delimited
No

Parameters
Bit[2] type
An enumerated type that indicates what type of link the color is being set for. One of:

- **cmlLinkColor**
  A link the user has not followed.

- **cmlLinkColorVisited**
  A link the user has followed previously.

- **cmlLinkColorActive**
  A link the user is tapping (the pen is down) at the moment. Once the pen is lifted, the color changes to the **visitedLinkColor**.

Byte red
A value from 0 to 255 that indicates the amount of red in the color.

Byte green
A value from 0 to 255 that indicates the amount of green in the color.

Byte blue
A value from 0 to 255 that indicates the amount of blue in the color.

Example
Tag tag = cmlTagLinkColor
Bit[2] type = cmlLinkColorVisited
Byte red = 0xFF
Byte green = 0x80
Byte blue = 0x80

cmlTagTextSize

Description
Sets the current text size.

End Tag Delimited
No

Parameters
Bit[3] size
HTML font size; a value from 1-7.
### cmlTagTextSize

**Description**
Marks text size.

**End Tag**
Delimited

**Parameters**
None

**Example**
```java
Tag tag = cmlTagTextSize
Bit[3] size = 3
```

### cmlTagTextBold

**Description**
Marks bold text style.

**End Tag**
Yes

**Parameters**
None

**Example**
```java
// Start bold text
Tag tag = cmlTagTextBold
Text "This is bold text"
// End bold text
Char end = cmlCharEnd
```

### cmlTagTextItalic

**Description**
Marks italic text style.

**End Tag**
Yes

**Parameters**
None

**Example**
```java
// Start italic text
Tag tag = cmlTagTextItalic
Text "This is italic text"
// End italic text
Char end = cmlCharEnd
```

### cmlTagTextStrike

**Description**
Marks strike-through text style.

**End Tag**
Yes

**Parameters**
None
Parameters None

Example
   // Start Strike-through text
   Tag tag = cmlTagTextStrike
   Text "This is strike-through text"
   // End strike-through text
   Char end = cmlCharEnd

**cmlTagTextMono**

Description Marks monospace text style.

End Tag Yes

Delimited Yes

Parameters None

Example
   // Start monospace text
   Tag tag = cmlTagTextMono
   Text "This is monospace text"
   // End monospace text
   Char end = cmlCharEnd

**cmlTagTextSup**

Description Marks superscript text style.

End Tag Yes

Delimited Yes

Parameters None

Example
   // Start superscript text
   Tag tag = cmlTagTextSup
   Text "This is superscript text"
   // End superscript text
   Char end = cmlCharEnd
cmlTagTextSub

Description  Marks subscript text style.

End Tag       Yes
Delimited     Yes
Parameters    None

Example      // Start subscript text
              Tag tag = cmlTagTextSub
              Text "This is subscript text"
              // End subscript text
              Char  end = cmlCharEnd

---

cmlTagTextUnderline

Description  Marks underlined text style.

End Tag       Yes
Delimited     Yes
Parameters    None

Example      // Start underlined text
              Tag tag = cmlTagTextUnderline
              Text "This is underlined text"
              // End underlined text
              Char  end = cmlCharEnd

---

cmlTag8BitEncoding

Description  Marks the beginning of 8-bit encoded text while in 5-bit encoding mode.

Note that in practice this tag is really a 5-bit encoding tag, and is only used within bit-packed data, not unpacked data.
End Tag		Yes

Delimited

Parameters	None

Example	Tag tag = cmlTag8BitEncoding
Text "THIS IS 8-BIT ENCODED TEXT"
// End 8-bit encoded text
Char end = cmlCharEnd

cmlTagH1, cmlTagH2, cmlTagH3, cmlTagH4, cmlTagH5, cmlTagH6

Description	Marks document headings.

End Tag		Yes

Delimited

Parameters	Bit hasAlign	A flag that is set if the align attribute is used.
If (hasAlign)
Bit[2] align	An enumerated type that sets how the heading is aligned horizontally in the window. One of {cmlAlignLeft, cmlAlignCenter, cmlAlignRight}

Example	Tag tag = cmlTagH1
Bit hasAlign = 1
Bit[2] align = alignCenter
Text "This is a Heading"
Char cmlCharEnd // end heading tag

cmlTagHistoryListText

Description	Transmits the content attribute of an HTML meta tag with the name attribute = “HistoryListText”. The value is stored as a NULL terminated string.
PQA Encoding Format
Tag Definitions

End Tag Delimited

No

Parameters

TextZ NULL terminated string value.

Example

Tag tag = cmlTagHistoryListText
TextZ "Portfolio&Date&Time"

Paragraph Attributes

cmlTagParagraphAlign

Description
Sets paragraph alignment.

End Tag Delimited

No

Parameters

Bit[2] align An enumerated type that sets how the paragraph is aligned horizontally in the window. One of {cmlAlignLeft, cmlAlignCenter, cmlAlignRight}

Example

// Turn on center alignment
Tag tag = cmlTagParagraphAlign
Bit[2] align = cmlAlignCenter
Text "\nThis paragraph is centered."
// Turn off center alignment
Tag tag = cmlTagParagraphAlign
Bit[2] align = cmlAlignLeft
Text "\nThis paragraph is left aligned."

cmlTagBlockQuote

Description
Delimits block quotations.

End Tag Delimited

Yes

Parameters

None
### Example

```
Tag tag = cmlTagBlockQuote
Text "The whole problem with the world is that fools and fanatics are always so certain of themselves, but wiser people so full of doubts."
Text "- Bertrand Russell"
Char cmlCharEnd // end block quote
```

### cmlTagAddress

**Description**

Delimits address data.

**End Tag**

Yes

**Parameters**

None

**Example**

```
Tag tag = cmlTagAddress
Text "Big Bird\nSesame St.\nNY, NY"
Char cmlCharEnd // end address
```

### Lists

#### cmlTagListOrdered

**Description**

Marks the beginning of an ordered (numbered) list of items. Each item in the list is preceded by either a `cmlTagListItemNormal` or `cmlTagListItemCustom` tag. A final `cmlCharEnd` character indicates the end of the list.

**End Tag**

Yes

**Parameters**

**Bit[3] type**

An enumerated type that indicates the type of numbering scheme. One of:

- `cmlListT1`
  - Counting numbers (1, 2, 3, ...)

---
cmlListTa
  Lowercase letters (a, b, c, ...)

cmlListTA
  Uppercase letters (A, B, C, ...)

cmlListTi
  Lowercase Roman numerals (i, ii, iii, ...)

cmlListTI
  Uppercase Roman numerals (I, II, III, ...)

Uint16V start  The starting sequence number, minus 1. (0 means start numbering with 1.)

Example
  // The list header
  Tag tag = cmlTagListOrdered
  Bit[3] type = cmlListT1
  Uint16V start = 0

  // The list items.
  Tag tag = cmlTagListItemNormal
  Text "First item"
  Tag tag = cmlTagListItemNormal
  Text "Second item"
  Tag tag = cmlTagListItemCustom
  Bit[2] mods = 0x03
  Bit[3] type = cmlListTa
  Uint16V value = 4
  Text "Third item"
  Char end = cmlCharEnd // end of list

**cmlTagListUnordered**

**Description**  Marks the beginning of an unordered list of items. Either a cmlTagListItemNormal or cmlTagListItemCustom tag precedes each item in the list. A final cmlCharEnd character indicates the end of the list.

**End Tag**  Yes

**Delimited**  Yes
Parameters

<table>
<thead>
<tr>
<th>Bit[3] type</th>
<th>An enumerated type that specifies the bullet type. One of:</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmlListTDisc</td>
<td>Filled circle bullet</td>
</tr>
<tr>
<td>cmlListTSquare</td>
<td>Filled square bullet</td>
</tr>
<tr>
<td>cmlListTCircle</td>
<td>Open circle bullet</td>
</tr>
</tbody>
</table>

Example

```cpp
// The list header
Tag tag = cmlTagListUnordered
Bit[3] type = cmlListTDisc
// The list items.
Tag tag = cmlTagListItemNormal
Text "First item"
Tag tag = cmlTagListItemNormal
Text "Second item"
Tag tag = cmlTagListItemCustom
Bit[2] mods = 0x01
Bit[3] type = cmlListTSquare
Text "Third item"
Char cmlCharEnd // end of list
```

cmlTagListDefinition

Description

Marks the beginning of a definition list. A `cmlTagListItemTerm` tag precedes each term and a `cmlTagListItemDefinition` precedes each definition. An `cmlCharEnd` character delimits the entire list.

End Tag Delimited

Yes

Parameters

None

Example

```cpp
Tag tag = cmlTagListDefinition

Tag tag = cmlTagListItemTerm
```
Text "This data corresponds to the first <DT> tag's data."
Tag tag = cmlTagListItemDefinition
Text "This data corresponds to the first <DD> tag's data."

Tag tag = cmlTagListItemTerm
Text "This data corresponds to the second <DT> tag's data."
Tag tag = cmlTagListItemDefinition
Text "This data corresponds to the second <DD> tag's data."

Char cmlCharEnd // end of list

cmlTagListItemNormal

Description Marks the beginning of a normal list item in either an ordered or unordered list. If the bullet style, numbering style, or sequence number of an item is not the default for the current list, the cmlTagListItemCustom tag must be used.

End Tag No

Delimited

Parameters None

Example Tag tag = cmlTagListItemNormal
Text "Third item"

cmlTagListItemCustom

Description Marks the beginning of a custom list item in either an ordered or unordered list. If the bullet style, numbering style, or sequence number of an item is not the default for the current list, this tag must be used.

The mods parameter indicates whether type, value, or both are specified.
End Tag
Delimited

Parameters

Bit[2] mods
Flags controlling these attributes:

cmlFlagListModValue[1]
Set if the value attribute is used.

cmlFlagListModType[0]
Set if the type attribute is used.

if (cmlFlagListModValue)

Uint16V value
Ignored for unordered lists. In ordered lists, value is the numeric value for this element, minus 1.

if (cmlFlagListModType)

Bit[3] type
The bullet or number style. An enumerated type. One of:

cmlListTDisc
Filled circle bullet

cmlListTSquare
Filled square bullet

cmlListTCircle
Open circle bullet

cmlListT1
Counting numbers (1, 2, 3, ...)

cmlListTa
Lowercase letters (a, b, c, ...)

cmlListTA
Uppercase letters (A, B, C, ...)

cmlListTi
Lowercase Roman numerals (i, ii, iii, ...)

cmlListTI
Uppercase Roman numerals (I, II, III, ...)

Example

Tag tag = cmlTagListItemCustom
Bit[2] mods = 0x03
Bit[3] type = cmlListTSquare
Uint16V value = 0
Text "Third item"

**cmlTagListItemTerm**

**Description**
Marks the beginning of a definition term in a definition list.

**End Tag**
No

**Delimited**
No

**Parameters**
None

**Example**
Tag tag = cmlTagListItemTerm
Text "Term for definition"

**cmlTagListItemDefinition**

**Description**
Marks the beginning of a definition of a term in a definition list.

**End Tag**
No

**Delimited**
No

**Parameters**
None

**Example**
Tag tag = cmlTagListItemDefinition
Text "Definition of term."

**Forms**

**cmlTagForm**

**Description**
Marks the start of a form. A form encloses one or more input items and is cmlCharEnd delimited.

There are essentially two classes of forms: stand-alone forms (like in standard HTML) and server dependent forms. Server dependent forms can be much smaller than standard forms and are typically the only type of form received over a wireless link. Stand-alone
forms, on the other hand, are designed to be contained within a PQA resident on the Palm device.

A stand-alone form is indicated by a 1 in the standalone attribute of the form tag. A 1 in this bit indicates that the form also has post and action attributes and that each of its input fields has the necessary attributes (name and value) for submitting the form without making the proxy reference the original HTML form on the Internet first.

A server dependent form is indicated by a 0 in the standalone attribute. A 0 in this bit indicates that the form does not have post or action attributes and that its input fields do not have associated name or value attributes. When this type of form is sent to the proxy server, the proxy server must first reference the original HTML form on the Internet before it can actually submit the request to the CGI script.

**End Tag**

Delimited: Yes

**Parameters**

Uint16V formIndex

Assigned by the proxy server; starts at 0 for the first form in a document.

Bit[3] flags

Flags controlling these attributes:

- cmflagFormIsLocalAction[2]
  
  Set when the protocol scheme identifies an action that is local to the device; that is, it is one of the set (file:, mailto:, palm:, palmcall:).

- cmflagFormIsSecure[1]
  
  Used only for server-dependent forms. Set if the action URL for the form is for a secure site (uses the https scheme). It is used by the client to determine if it should send the form submission to the proxy in encrypted form or not. For stand-alone forms, the client should instead check the scheme that’s in the
PQA Encoding Format
Tag Definitions

action URL parameter so see if the submission should be encrypted or not.

cmlFlagFormIsStandalone[0]
Set if the form is stand-alone; not set if the form is server dependent.

if(cmlFlagFormIsStandalone)

Bit post If set to 1, the form is submitted to the CGI script using the HTTP POST method; if set to 0, the form is submitted to the CGI script using the HTTP GET method.

TextZ encType String that specifies the type of form encoding. If no format is specified in the HTML, then this string is NULL and the default, “application/x-www-form-urlencoded” is implied.

TextZ action URL of the CGI script on the server that handles the form submission.

Example
Tag tag = cmlTagForm
Uint16V formIndex = 0
Bit[3] flags = 1 // cmlFlagFormIsStandalone
Bit post = 0
TextZ encType = 0
TextZ action = "http://www.server.com/cgi-bin/submit"

// The form input items
Text "Age 0-12:"
Tag tag = cmlTagInputRadio
Uint16V group = 0
Bit[4] flags = 3 // has name, value
TextZ name = "age"
TextZ value = "0-12"

Text "Age 13-17:"
Tag tag = cmlTagInputRadio
Uint16V group = 0
Bit [4] flags = 7 // has name, value, is checked
TextZ name = "age"
TextZ value = "13-17"

Tag tag = cmlTagInputSubmit
Bit[2] flags = 2 // has value
TextZ value = "OK"

Char endForm = cmlCharEnd

**cmlTagInputTextLine**

**Description**
Marks a single line input text field in a form.

**End Tag**
No

**Delimited**
No

**Parameters**

- Uint16V size
  Visible width of field in characters.
- Uint16V maxLength
  Maximum number of allowed characters. 0 means no limit.
- Bit[2] flags
  Flags controlling these attributes:
  - cmlFlagInputHasValue[1]
    Set if the hasValue attribute is used.
  - cmlFlagInputHasName[0]
    Set if the hasName attribute is used. Set only in stand-alone forms.

  if(cmlFlagInputHasName)
  TextZ name
  String holding the name of the input field.

  if(cmlFlagInputHasValue)
  TextZ value
  String holding the initial value for the input field.

**Example**

Tag tag = cmlTagForm
Text "Enter Name:"

Tag tag = cmlTagInputTextLine
PQA Encoding Format
Tag Definitions

Uint16V size = 20
Uint16V maxLength = 0
Bit[2] flags = 3
TextZ name = "name"
TextZ value = "your name here"

cmlTagInputPassword

Description Marks a single line password input field in a form.

End Tag Delimited No

Parameters

Uint16V size Visible width of field in characters.
Uint16V maxLength Maximum number of allowed characters. Specify 0 for no limit.
Bit[2] flags Flags controlling these attributes:
   cmlFlagInputHasValue[1] Set if the hasValue attribute is used.
   cmlFlagInputHasName[0] Set if the hasName attribute is used. Set only in stand-alone forms.

if(cmlFlagInputHasName)
TextZ name String holding the name of the input field.

if(cmlFlagInputHasValue)
TextZ value String holding the initial value for the input field.

Example
Text "Enter Password:"
Tag tag = cmlTagInputPassword
Uint16V size = 20
Uint16V maxLength = 0
Bit[2] flags = 1
TextZ name = "passwd"
**cmlTagInputRadio**

**Description**
Marks a radio button in a form.

**End Tag Delimited**
No

**Parameters**

- **Uint16 group**
  Assigned by the proxy server; it allows the client to perform mutual exclusion selecting.

- **Bit[4] flags**
  Flags controlling these attributes:
  
  - `cmlFlagInputHasText[3]`
    Set if the `Text` attribute is included as an active part of the radio button; that is, in Clipper, the user can tap the text as well as the button to operate the control. The encoder automatically sets this bit for HTML pages that are identified by the PalmComputingPlatform meta tag. If this bit is not set, then the radio button label appears as a separate text string before or after the radio button tag.

  - `cmlFlagInputChecked[2]`
    Indicates the initial state of the control. If set, the control is selected.

  - `cmlFlagInputHasValue[1]`
    Set if the `hasValue` attribute is used. If this attribute is not used, the string “on” is sent to the server if the control is selected.

  - `cmlFlagInputHasName[0]`
    Set if the `hasName` attribute is used. Set only in stand-alone forms.

**CML Tags**

- **if (cmlFlagInputHasName)**
  
  **TextZ** name
  String holding the name of the radio button control.

- **if (cmlFlagInputHasValue)**
PQA Encoding Format

Tag Definitions

TextZ value  String holding the value for the radio button. This value is sent to the server if the control is selected.

if(cmlFlagInputHasText)
  TextZ Text  String holding the text label next to the control. This label is included as an active part of the radio button.

Example
Tag tag = cmlTagInputRadio
Uint16V group = 0
Bit[4] flags = 0xB // cmlFlagInputHasName | cmlFlagInputHasValue | cmlFlagInputHasText
TextZ name = "age"
TextZ value = "13-17"
TextZ Text = "Age 13-17:"

cmlTagInputCheckBox

Description  Marks a checkbox in a form.

End Tag Delimited  No

Parameters  Bit[4] flags  Flags controlling these attributes:
  cmlFlagInputHasText[3]
  Set if the Text attribute is included as an active part of the checkbox; that is, in Clipper, the user can tap the text as well as the checkbox to operate the control. The encoder automatically sets this bit for HTML pages that are identified by the PalmComputingPlatform meta tag. If this bit is not set, then the checkbox label appears as a separate text string before or after the checkbox tag.
  cmlFlagInputChecked[2]
  Indicates the initial state of the control. If set, the control is checked.
PQA Encoding Format
Tag Definitions

**cmlFlagInputHasValue[1]**
Set if the `hasValue` attribute is used. If this attribute is not used, the string “on” is sent to the server if the control is selected.

**cmlFlagInputHasName[0]**
Set if the `hasName` attribute is used. Set only in stand-alone forms.

```java
if (cmlFlagInputHasName)
    TextZ name    String specifying the name of the checkbox.
```  
```java
if (cmlFlagInputHasValue)
    TextZ value   String holding the value for the checkbox. This value is sent to the server if the control is selected.
```  
```java
if (cmlFlagInputHasText)
    TextZ Text    String holding the text label next to the control. This label is included as an active part of the checkbox.
```

**Example**
```
Tag tag = cmlTagInputCheckBox
Bit[4] flags = 3 // cmlFlagInputHasName | cmlFlagInputHasValue
TextZ name = "newsletter"
TextZ value = "1"

// Checkbox label is not part of the object. // It could be formatted text or an image.
Text "Yes" // checkbox label, not active
```

**cmlTagInputSubmit**

**Description**
Marks a submit button in a form.

**End Tag**
No

**Delimited**
No

**Parameters**
Bit[2] flags Flags controlling these attributes:
PQA Encoding Format
Tag Definitions

Palm File Formats Specification
modified 1/17/00 Preliminary

```
cmlFlagInputHasValue[1]
    Set if the hasValue attribute is used to set a custom button label.

cmlFlagInputHasName[0]
    Set if the hasName attribute is used. Set only in stand-alone forms.
```

```
if (cmlFlagInputHasName)
    TextZ name       String holding the name of the button.

if (cmlFlagInputHasValue)
    TextZ value      String holding the button label. If this parameter is not included, the default button label is “submit.”
```

**Example**
```
Tag tag = cmlTagInputSubmit
Bit[2] flags = 2
TextZ value = "OK"
```

**cmlTagInputReset**

**Description**
Marks a reset button in a form.

**End Tag**
No

**Delimited**
No

**Parameters**

```
Bit hasValue       Set if the hasValue attribute is used to set a custom button label.
```

```
if (hasValue)
    TextZ value      String holding the button label. If this parameter is not included, the default button label is “reset.”
```

**Example**
```
Tag tag = cmlTagInputReset
Bit hasValue = 1
TextZ value = "Clear Form"
```
**cmlTagInputHidden**

**Description**
Marks a hidden input field in a form. This tag is not generated for server supplied forms except for value strings of either “%zipcode” or “%deviceid”.

**End Tag**
No

**Delimited**
No

**Parameters**
- **Bit[2] flags**
  Flags controlling these attributes:
  - cmlFlagInputHasValue[1]
    Set if the hasValue attribute is used to set a custom button label.
  - cmlFlagInputHasName[0]
    Set if the hasName attribute is used. Set only in stand-alone forms.

  if (cmlFlagInputHasName)
  TextZ name
   String holding the name of the input field.
  
  if (cmlFlagInputHasValue)
  TextZ value
   String holding the initial value for the input field.

**Example**
```javascript
Tag tag = cmlTagInputHidden
Bit[2] flags = 3
TextZ name = "Age"
TextZ value = "21"
```

**cmlTagInputTextArea**

**Description**
Marks a multi-line input text field within a form.

**End Tag**
Yes

**Delimited**
Yes

**Parameters**
- **Uint16V rows**
  Number of rows in the input field.
- **Uint16V cols**
  Width of the input field in characters.
Bit hasName
Set if the hasName attribute is used to set an input field name. Set only in stand-alone forms.

if (hasName)
  TextZ name
  String holding the name of the input field.
  TextZ value
  String holding the initial value for the input field. The end of the initial text is indicated by a cmlCharEnd character.

Example
Text "Enter Address:"
Tag tag = cmlTagInputTextArea
Uint16V rows = 2
Uint16V cols = 20
Bit hasName = 1
TextZ name = "address"
TextZ value = "your address 
here: "
Char cmlCharEnd

cmlTagSelect

Description
Marks a selection menu in a form.

This element is always followed by one or more TextZ elements that represent the menu items; these are separated by cmlTagSelectItemNormal or cmlTagSelectItemCustom tags. The cmlTagSelectItemCustom tag is used for preselected items. A cmlCharEnd character follows the last item and indicates the end of the selection menu.

End Tag
Delimited
Yes

Parameters
Bit[2] flags
Flags controlling these attributes:
cmlFlagInputMultiple[1]
  Set if multiple item selection is allowed.
cmlFlagInputHasName[0]
  Set if the hasName attribute is used. Set only in stand-alone forms.
Uint16V size    Number of items visible at once in the selection list, minus 1.

if (cmlFlagInputHasName)
  TextZ name    String holding the name of the selection menu.

Example
  Tag tag = cmlTagSelect
  Bit[2] flags = 3
  Uint16V size = 2
  TextZ name = "choice"

  // The select items.
  Tag tag = cmlTagSelectItemNormal
  TextZ "First choice"
  Tag tag = cmlTagSelectItemCustom
  Bit[2] flags = 1
  TextZ "Second choice"
  Tag tag = cmlTagSelectItemNormal
  TextZ "Third choice"

  Char endSelect = cmlCharEnd

\textbf{cmlTagSelectItemNormal}
\begin{description}
\item[Description] Precedes a normal item in a selection menu. A normal item means that it is not preselected and it does not have a value different from its text content.
\item[End Tag] No
\item[Delimited] No
\item[Parameters] None
\item[Example] Tag tag = cmlTagSelectItemNormal
  TextZ "Third item"
\end{description}

\textbf{cmlTagSelectItemCustom}
\begin{description}
\item[Description] Precedes a custom item in a selection menu.
\end{description}
**cmlTagSelectItemCustom**

**Parameters**

- **Bit[2] flags**
  Flags controlling these attributes:
  - `cmlFlagInputHasValue[1]` Set if the `hasValue` attribute is used. Set only in stand-alone forms.
  - `cmlFlagInputSelected[0]` Set if the item is to be preselected in the menu.

  ```
  if(cmlFlagInputHasValue)
  TextZ value A string holding text that should be used as the value of this item at form submission. If this parameter is omitted, then the TextZ string that follows the cmlTagSelectItemCustom tag is used instead.
  ```

**Example**

```java
Tag tag = cmlTagSelectItemCustom
Bit[2] flags = 3
TextZ value = "3"
TextZ "Third item"
```

Mark a date picker.

**cmlTagInputDatePicker**

**Parameters**

- **Bit hasName**
  Set if the `name` attribute is used to set a name for the date field.

- **UIntV date**
  The initial value of the date field; the number of seconds since midnight, 1/1/1904 GMT. Specify 0 to use the current date.

  ```
  if(hasName)
  TextZ name String holding the name of the date field.
  ```
Example

Tag tag = cmlTagInputDatePicker
Bit hasName = 1
UIntV date = 0xA1234000
TextZ name = "yesterday"

**cmlTagInputTimePicker**

**Description**
Marks a time picker.

**End Tag**
No

**Delimited**
No

**Parameters**
- **Bit hasName**
  Set if the `name` attribute is used to set a name for the time field.
- **UIntV seconds**
The initial value of the time field; the number of seconds since midnight. Specify 0 to use the current time.

```java
if (hasName)
  TextZ name
String holding the name of the time field.
```

Example

Tag tag = cmlTagInputTimePicker
Bit hasName = 0
UIntV seconds = 3600 // 1:00 am

**Tables**

**cmlTagTable**

**Description**
Indicates the start of a table.

Each row in the table begins with a `cmlTagTableRow` tag that has optional parameters for the horizontal and vertical alignment of the cells in that row.

Each cell in a row begins with either a `cmlTagTableData` or a `cmlTagTableHeader` tag. The only difference is that header cells are rendered in bold typeface. After the last row, an additional `cmlCharEnd` indicates the end of the table.
End Tag
Delimited

Parameters

Flags controlling these attributes:

- **cmlFlagTableHasAlign[0]**
  Set if the hAlign attribute is used.

- **cmlFlagTableHasWidth[1]**
  Set if the width attribute is used.

- **cmlFlagTableHasBorder[2]**
  Set if the border attribute is used.

- **cmlFlagTableHasCellSpacing[3]**
  Set if the cellSpacing attribute is used.

- **cmlFlagTableHasCellPadding[4]**
  Set if the cellPadding attribute is used.

- **reserved1[5]**
  Not used.

- **reserved2[6]**
  Not used.

If (cmlFlagTableHasAlign)

- **Bit[2] hAlign**
  An enumerated type setting how the table is aligned on the page. One of `{cmlAlignLeft, cmlAlignCenter, cmlAlignRight}`.

If (cmlFlagTableHasWidth)

- **Uint16V width**
  Table width in pixels. 0 indicates to calculate the width of the table from the contents.

If (cmlFlagTableHasBorder)

- **Uint8V border**
  Border width in pixels. 0 indicates to suppress the border.

If (cmlFlagTableHasCellSpacing)

- **Uint8V cellSpacing**
  Cell spacing in pixels. The cell spacing is the distance between the borders of each cell. If non-zero, then cells are spaced apart from each other. The default is 0.
If (cmlFlagTableHasCellPadding)
Uint8V cellPadding

Cell padding in pixels. The cell padding is the distance between the border around each cell and the cell's contents. The default is 0.

Example
Tag tag = cmlTagTable
Bit[7] flags = 0x01 // cmlFlagTableHasAlign
Bit[2] hAlign = cmlAlignCenter

Tag tag = cmlTagTableRow
Bit hasAlign = 0
Tag tag = cmlTagTableHeader
Bit[7] flags = 0
Text "Row1, Col2 Head"
Char cmlCharEnd
Tag tag = cmlTagTableHeader
Bit[7] flags = 0
Text "Row1, Col2 Head"
Char cmlCharEnd

Tag tag = cmlTagTableRow
Bit hasAlign = 0
Tag tag = cmlTagTableData
Bit[7] flags = 0
Text "row2, col1"
Char cmlCharEnd
Tag tag = cmlTagTableData
Bit[7] flags = 0
Text "row2, col2"
Char cmlCharEnd

Char cmlCharEnd // end of table

cmlTagCaption

Description  Marks the caption to be placed above or below a table. It can appear anywhere in a table.
PQA Encoding Format
Tag Definitions

**End Tag**
Delimited Yes

**Parameters**
Bit captionAtTop
A Boolean value. 0 means place the caption below the table; 1 means place the caption above the table.

**Example**
Tag tag = cmlTagCaption
Bit captionAtTop = 1
Text "Table Title"
Char cmlCharEnd // end of caption

**cmlTagTableRow**

**Description**
Separates rows of a table.
Each row in the table begins with a cmlTagTableRow tag.

**End Tag**
Delimited Yes

**Parameters**
Bit hasAlign
Set if the hAlign and vAlign attributes are used.

if(hasAlign)

Bit[2] hAlign
An enumerated type that sets how text is aligned horizontally within the cells in the row.
One of {cmlAlignLeft, cmlAlignCenter, cmlAlignRight}.

Bit[2] vAlign
An enumerated type that sets how text is aligned vertically within the cells in the row.
One of {cmlVAlignTop, cmlVAlignCenter, cmlVAlignBottom}.

**Example**
Tag tag = cmlTagTableRow
Bit hasAlign = 0
Tag tag = cmlTagTableHeader
Bit[7] flags = 0
Text "row1, col1"
cmlTagTableData

Description Marks a data cell in a table. Contrast this tag with cmlTagTableHeader.

End Tag Delimited Yes

Parameters Bit[7] flags Flags controlling these attributes:

  cmlFlagCellHasHAlign[0] Set if the hAlign attribute is used.

  cmlFlagCellHasVAlign[1] Set if the vAlign attribute is used.

  cmlFlagCellHasColSpan[2] Set if the colSpan attribute is used.

  cmlFlagCellHasRowSpan[3] Set if the rowSpan attribute is used.

  cmlFlagCellHasHeight[4] Set if the height attribute is used.

  cmlFlagCellHasWidth[5] Set if the width attribute is used.

  cmlFlagCellNoWrap[6] Set if automatic word wrap within the contents of the cell is disabled.

Parameters If(cmlFlagCellHasHAlign)
### PQA Encoding Format

#### Tag Definitions

- **Bit[2] hAlign**: An enumerated type that sets horizontal cell alignment. One of \{cmlAlignLeft, cmlAlignCenter, cmlAlignRight\}.

  - **If** (cmlFlagCellHasVAlign)
    - **Bit[2] vAlign**: An enumerated type that sets vertical cell alignment. One of \{cmlVAAlignTop, cmlVAAlignCenter, cmlVAAlignBottom\}.

  - **If** (cmlFlagCellHasColSpan)
    - **Uint8V colSpan**: Number of columns spanned by the cell, minus 1. For example, if the cell spans one column, this is set to 0.

  - **If** (cmlFlagCellHasRowSpan)
    - **Uint8V rowSpan**: Number of rows spanned by the cell, minus 1.

  - **If** (cmlFlagCellHasHeight)
    - **Uint16V height**: Height of the cell in pixels.

  - **If** (cmlFlagCellHasWidth)
    - **Uint16V width**: Width of the cell in pixels.

#### Example

```
Tag tag = cmlTagTableData
Bit[7] flags = 0x11 // cmlFlagCellHasColSpan | cmlFlagCellHasHAlign
Uint8V colSpan = 1
Bit[2] hAlign = cmlAlignCenter
Text "row2, col2"
Char cmlCharEnd
```

### cmlTagTableHeader

#### Description

Marks a header cell in a table. Header cells are rendered in bold typeface. Contrast this tag with `cmlTagTableData`. 
End Tag  
Delimited  
Yes

Parameters  
Bit[7] flags  
Flags controlling these attributes:

- cmlFlagCellHasHAlign[0]  
  Set if the hAlign attribute is used.

- cmlFlagCellHasVAlign[1]  
  Set if the vAlign attribute is used.

- cmlFlagCellHasColSpan[2]  
  Set if the colSpan attribute is used.

- cmlFlagCellHasRowSpan[3]  
  Set if the rowSpan attribute is used.

- cmlFlagCellHasHeight[4]  
  Set if the height attribute is used.

- cmlFlagCellHasWidth[5]  
  Set if the width attribute is used.

- cmlFlagCellNoWrap[6]  
  Set if automatic word wrap within the contents of the cell is disabled.

Parameters  
If(cmlFlagCellHasHAlign)

- Bit[2] hAlign  
  An enumerated type that sets horizontal cell alignment. One of {cmlAlignLeft, cmlAlignCenter, cmlAlignRight}.

Parameters  
If(cmlFlagCellHasVAlign)

- Bit[2] vAlign  
  An enumerated type that sets vertical cell alignment. One of {cmlVAlignTop, cmlVAlignCenter, cmlVAlignBottom}.

Parameters  
If(cmlFlagCellHasColSpan)

- Uint8V colSpan  
  Number of columns spanned by the cell, minus 1. For example, if the cell spans one column, this is set to 0.

Parameters  
If(cmlFlagCellHasRowSpan)
Uint8V rowSpan
    Number of rows spanned by the cell, minus 1.

If (cmlFlagCellHasHeight)
    Uint16V height
    Height of the cell in pixels.

If (cmlFlagCellHasWidth)
    Uint16V width  Width of the cell in pixels.

Example

    Tag tag = cmlTagTableHeader
    Bit[7] flags = 0x14  // cmlFlagCellHasColSpan | cmlFlagCellHasHeight
    Uint8V colSpan = 1
    Uint16V height = 10
    Text "row1, col2"
    Char cmlCharEnd

Hyperlinks

cmlTagHyperlink

Description  Marks a hyperlink. All text enclosed between the
cmlTagHyperlink tag and the terminating cmlCharEnd is part of
the hyperlink.

Unlike the anchor (<A>) element in HTML, which can be used to
define both hyperlinks and named anchors (that is, fragment
identifiers using the NAME attribute), the cmlTagHyperlink tag
is used only to define hyperlinks. The cmlTagAnchor tag, defined
below, is used exclusively to define named anchors.

End Tag  Yes

Delimited  Yes

Parameters

    Bit[2] flags2  Flags controlling these attributes:
    cmlFlagLinkIsBinary[1]
        Not currently used.
cmlFlagLinkIsLocalRef[0]
Set if this hyperlink’s URL specifies a
device-side scheme (e.g. file:).

Bit[8] flags Flags controlling these attributes:

- cmlFlagLinkIsFakeRemote[7]
  Set if this hyperlink is used by the
  Palm OS and is set to simulate a wireless
  request by delaying access to the
  (hopefully) internal data.

- cmlFlagLinkIsSameDoc[6]
  Set if this is a hyperlink into the current
document.

- cmlFlagLinkHasHref[5]
  Set if an hRef attribute is included. If
  hasHref is false, then the
  extLinkIndex and hashValue
  attributes are provided. In this case, the
  data was probably received via the server
  and the enumeration of hyperlinks
  present in the current file must be used in
  the data request.¹

- cmlFlagLinkIsSecure[4]
  Set if the hyperlink is to a secure page.

- cmlFlagLinkIsFragment[3]
  Set if the hyperlink references a fragment
  within the same page; the
  fragmentName attribute is provided.

- cmlFlagLinkInternal[2]
  Set if this hyperlink references a
document in the current PQA file. In this
  case, the PQFIndex attribute is
  provided. If internal is false, then a

¹Normally, in pages received from the Palm Web Clipping Proxy server, all hyperlink
URLs are removed before the page is sent to the Palm VII unit. If the user taps a hy-
perlink, the index of that link is sent back to the server, which refetches the page and
finds the corresponding URL.
PQA Encoding Format
Tag Definitions

- complete representation of the URL is provided if the hasHref bit is true.

  cmlFlagLinkHasTitle[1]
  Set if a title attribute is included.

  cmlFlagLinkIsButton[0]
  Set if this hyperlink should be displayed as a button rather than text.

if (cmlFlagLinkIsSameDoc)
if (cmlFlagLinkIsUnicode)
  TextZ fragmentName
  String holding the fragment portion of the URL. For example, if the URL is "file:\foo.htm#section1", then the fragment is "section1".

else if (cmlFlagLinkIsExternal)
  Uint16V PQFIndex
  The index of the resource (referenced by the hyperlink) in the current PQA file. The first resource has an index of 1.

If (cmlFlagLinkIsFragment)
  TextZ fragmentName
  String holding the fragment portion of the URL.

else // cmlFlagLinkIsExternal is false
If (!cmlFlagLinkHasHref) // cmlFlagLinkHasHref is false
  Uint16V extLinkIndex
  The index of the link on the page. This is used only for external links from external (non-PQA) pages.

  Uint16V hashValue
  A hash value for the page that is used to check if the page source has changed when it is refetched to retrieve a URL. (See the previous footnote.) This is used only for external links from external (non-PQA) pages.
else // cmlFlagLinkHasHref is true
    TextZ href    String holding the complete URL.
if(cmlFlagLinkHasTitle)
    TextZ title    String holding the title of the referenced page.

Example
Here is an example of an external explicit link that would typically be used by a document designed to be loaded onto a Palm device through the HotSync mechanism or some other non-wireless means:

Tag tag = cmlTagHyperlink
Bit[2] flags 2 = 0
Bit[8] flags = 0x22  // cmlFlagLinkHasTitle, cmlFlagLinkHasHref
TextZ href = "http://www.Palm.com/
TextZ title = "Palm home page"
Text "Click on this text"
Char cmlCharEnd // terminates cmlTagHyperlink

Here is an example of an external indexed link that would typically be used by a document that was obtained from a wireless link. Notice that it does not include a URL or a title in order to conserve space.

Tag tag = cmlTagHyperlink
Bit[2] flags2 = 0
Bit[8] flags = 0
Uint16V extLinkIndex = 14
Uint16V hashValue = 3056
Text "Click on this text"
Char cmlCharEnd // terminates cmlTagHyperlink

Here is an example of an internal link that is used to jump to another document within the same PQA. It indicates to jump to the fourth resource in the current PQA file.

Tag tag = cmlTagHyperlink
Bit[2] flags2 = 0
Bit[6] flags = 4  // cmlFlagLinkInternal
Uint16V PQFIndex = 4
Text "Click on this text"
Char cmlCharEnd // terminates cmlTagHyperlink
PQA Encoding Format

Tag Definitions

**cmlTagAnchor**

**Description**
Marks a named document anchor, or fragment identifier, within a document.

The `cmlTagAnchor` tag is used only to define local named anchors. The `cmlTagHyperlink` tag is used exclusively to define hyperlinks.

**End Tag**
Yes

**Delimited**
Yes

**Parameters**
- `TextZ name` String holding the local anchor name (not including the preceding “#” character).

**Example**
```java
Tag tag = cmlTagAnchor
TextZ name = "anchor"
```

**Graphical Elements**

**cmlTagImage**

**Description**
Marks an image.

**End Tag**
No

**Delimited**
No

**Parameters**
- `Bit[8] flags` Flags controlling these attributes:
  - `cmlFlagImageLocalPQA[7]` Set if the image is a resource in the current PQA.
  - `cmlFlagImageHasAlt[6]` Set if an `alt` attribute is included.
  - `cmlFlagImageHasSrc[5]` Set if a `src` attribute is included.
  - `cmlFlagImageHasVSpace[4]` Set if a `vSpace` attribute is included.
cmlFlagImageHasHSpace[3]
Set if an hSpace attribute is included.

cmlFlagImageHasBorder[2]
Set if a border attribute is included.

cmlFlagImageHasAlign[1]
Set if an align attribute is included.

cmlFlagImageEmbedded[0]
Set if the image is embedded into the data stream received from the Palm Web Clipping Proxy server. The image data is included in the imageData attribute.

If (cmlFlagImageLocalPQA)
Uint16V PQFLinkIndex
The index of the image resource in the current PQA file. The first resource has an index of 1.

if (cmlFlagImageHasAlt)
TextZ alt
Alternate text string for the image.

if (cmlFlagImageHasSrc)
TextZ src
Source URL; only for references to resources in other (than the current) PQA files.

if (cmlFlagImageHasVSpace)
Uint8V vSpace
Vertical space between the image and the text above and below, in pixels, minus 1.

if (cmlFlagImageHasHSpace)
Uint8V hSpace
Horizontal space between the image and the text to the left and right, in pixels, minus 1.

if (cmlFlagImageHasBorder)
Uint8V border
Border width in pixels, minus 1.

If (cmlFlagImageHasAlign)
Bit[3] align
An enumerated type that sets how the image is aligned relative to the text. One of:
PQA Encoding Format
Tag Definitions

```
cmlIAlignLeft
  Image is aligned to left side of window,
  and subsequent text wraps around right
  side of image. Creates a “floating” image.

cmlIAlignRight
  Image is aligned to right side of window,
  and subsequent text wraps around left
  side of image. Creates a “floating” image.

cmlIAlignTop
  Subsequent text is aligned to the top of
  the image.

cmlIAlignMiddle
  Baseline of the current text line is aligned
  with the middle of the image.

cmlIAlignBottom
  Bottom of the image is aligned with the
  baseline of the current text line.
```

If (cmlFlagImageEmbedded)

```
Image imageData
  Image data in Palm OS bitmap format.
```

Example
```
Tag tag = cmlTagImage
Bit[8] flags = 0x01 // IsEmbedded
Image imageData = //image data stream

Tag tag = cmlTagImage
Bit[8] flags = 0x86 // cmlFlagImageHasAlign,
  cmlFlagImageHasBorder, cmlFlagImageLocalPQA
Uint16V PQFLinkIndex = 4
Bit[3] Align = cmlIAlignTop
Uint8V Border = 3 // border of 4 pixels
```

cmlTagHorizontalRule

Description
Places a horizontal rule graphic in the window. If no attributes are
specified, the default rule appearance is set by the Clipper.
application. However, if one or more attributes are specified, the
defaults listed below apply (which may be different from Clipper).

End Tag
Delimited

No

Parameters Bit[5] flags Flags controlling these attributes:

  - cmlFlagHRIsPercent[4]
    Set if the percent or width attributes are included to specify rule width. The default is true.

  - cmlFlagHRNoShade[3]
    Set if the rule is not shaded. Not set if the rule is shaded.

  - cmlFlagHRAppear[2-1]
    An enumerated type that sets how the rule is horizontally aligned if it is less than the full width of the window. One of {cmlAlignLeft, cmlAlignCenter, cmlAlignRight}.

  - cmlFlagHRCustom[0]
    Set if other parameters are used. If not set, this indicates that no other parameters follow and a default rule is used, as determined by the Clipper application.

if (cmlFlagHRCustom)
  Uint16V size Height (thickness) of the rule in pixels. The default is 1.

if (cmlFlagHRIsPercent)
  Byte percent Relative width of the rule in percentage of display width. The default is 100.

else
  Uint16V width Absolute width of the rule in pixels.
Example

// A default rule
Tag tag = cmlTagHorizontalRule
Bit[5] flags = 0
Text "Some random text"

// A custom rule
Tag tag = cmlTagHorizontalRule
Bit[5] flags = 0x13 // cmlFlagHRCustom, cmlAlignCenter, cmlFlagHRIsPercent
Uint16V size = 3
Byte percent = 20

Other Elements

cmlTagClear

Description
Indicates that the browser should insert a line break and avoid floating images before continuing to draw text. Corresponds to the HTML element <BR CLEAR>.

End Tag
No

Delimited
No

Parameters
Bit[2] clearAlign
An enumerated type. One of:

- cmlClearLeft
  Break the line, and move vertically down until there is a clear left margin (where there are no floating images).

- cmlClearAll
  Break the line, and move vertically down until there is a clear right margin (where there are no floating images).

- cmlClearRight
  Break the line, and move vertically down until both margins are clear of images.
Example
Tag tag = cmlTagClear
Bit[2] clearAlign = cmlClearAll

cmlTagCMLEnd

Description
Indicates the end of data for this resource.

End Tag
No

Delimited
No

Parameters
None

Example
Tag tag = cmlTagCMLEnd
PQA Encoding Format
Tag Definitions
Index

Numerics
5-bit special characters 25
8-bit encoding tag 46

A
address tag 49
alignment of paragraphs 48
anchor tag 74
anchor, named 78
appInfo block 8
ASCII encoding 28

B
background color 41
bit packed compression 24
ASCII encoding 28
image encoding 29
numeric parameter encoding 29
tag encoding 28
translating to uncompressed data 38
block quote tag 48
bold text 44

C
checkbox 60
clear tag 82
cmlCompressionTypeBitPacked 18
cmlCompressionTypeNone 18
cmlContentTypeImagePalmOS 18
cmlContentTypeTextCml 18
cmlTag8BitEncoding 46
cmlTagAddress 49
cmlTagAnchor 78
cmlTagBGColor 41
cmlTagBlockQuote 48
cmlTagCaption 69
cmlTagClear 82
cmlTagCMLEnd 41, 83
cmlTagForm 54
cmlTagH1 47
cmlTagH2 47
cmlTagH3 47
cmlTagH4 47

cmlTagH5 47
cmlTagH6 47
cmlTagHistoryListText 47
cmlTagHorizontalRule 80
cmlTagHyperlink 74
cmlTagImage 78
cmlTagInputCheckBox 60
cmlTagInputDatepicker 66
cmlTagInputHidden 63
cmlTagInputPassword 58
cmlTagInputRadioButton 59
cmlTagInputReset 62
cmlTagInputSubmit 61
cmlTagInputTextArea 63
cmlTagInputTextLine 57
cmlTagInputTimePicker 67
cmlTagLinkColor 42
cmlTagListDefinition 51
cmlTagListItemCustom 52
cmlTagListItemDefinition 54
cmlTagListItemNormal 52
cmlTagListTerm 54
cmlTagListOrdered 49
cmlTagListUnordered 50
cmlTagParagraphAlign 48
cmlTagSelect 64
cmlTagSelectItemCustom 65
cmlTagSelectItemNormal 65
cmlTagTable 67
cmlTagTableData 71
cmlTagTableHeader 72
cmlTagTableRow 70
cmlTagTextBold 44
cmlTagTextColor 42
cmlTagTextItalic 44
cmlTagTextMono 45
cmlTagTextSize 43
cmlTagTextStrike 44
cmlTagTextSub 46
cmlTagTextSup 45
cmlTagTextUnderline 46

color
background 41
link 42
Index

- text 42
- compact data structure notation 29
  - data types 30
- compressed PQA data 24
- compression types 17
- content types 17

D
- data cell of table 71
- data termination 41
- database format 2
- DatabaseHdrType 3
- date picker 66
- definition list start 51
- definition, list 54

E
- encoding format 21

F
- form
  - checkbox 60
  - date picker 66
  - hidden field 63
  - password input line 58
  - radio button 59
  - reset button 62
  - selection item, custom 65
  - selection item, normal 65
  - selection menu 64
  - submit button 61
  - text area 63
  - text input line 57
  - time picker 67
- form tag 54
- format of PQA 21

G
- graphic tag 78

H
- header cell of table 72
- heading tags 47

hidden field 63
- history list tag 47
- horizontal rule 80
- HTML differences 23
- HTML text content 18
- hyperlink tag 74

I
- image content 18
- image encoding 29
- image tag 78
- input line 57
- Int16V 31
- Int8V 32
- IntV 31
- italic text 44

L
- link color 42
- list
  - custom item 52
  - definition in 54
  - definition start 51
  - normal item 52
  - ordered 49
  - term 54
  - unordered 50

M
- monospace text 45

N
- named anchor 78
- numeric parameter encoding 29

O
- ordered list 49

P
- paragraph alignment 48
- password input line 58
- PDB header 3
  - record list 7
Index

plain text content 18
PQA database format 2
PQA encoding format 21
PQA tags
definitions 41
overview 32
PqfWebDocRecordType 11

Q
quote, block 48

R
radio button 59
record list in PDB header 7
reset button 62
row, table 70
rule 80

S
selection menu 64
special 5-bit characters 25
strike-through text 44
submit button 61
subscript text 46
superscript text 45

T
table 67
data cell 71
header cell 72
row 70
table caption 69
tag
data type 35
definitions 41
encoding 28
PQA overview 32
text encoding 33
term, list 54
termination of data 41, 83
text
bold 44
color 42
encoding tags 33
italic 44
monospace 45
size 43
strike-through 44
subscript 46
superscript 45
underline 46
text area 63
Text type 35
TextZ type 36
time picker 67
translation of bit-packed to uncompressed data 38

U
Uint16V 31
Uint8V 31
UIntV 30
uncompressed notation 37
underline text 46
unordered list 50
unpacked notation 37

W
web content record 11