Palm® File Format Specification

CONTRIBUTORS

Written by Gary Hillerson
Production by `<dot>PS` Document Production Services
Engineering contributions by Kenneth Albanowski, John Marshall, Keith Rollin

Copyright © 1996 - 2001, Palm, Inc. All rights reserved. This documentation may be printed and copied solely for use in developing products for Palm OS software. In addition, two (2) copies of this documentation may be made for archival and backup purposes. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form or by any means or used to make any derivative work (such as translation, transformation or adaptation) without express written consent from Palm, Inc.

Palm, Inc. reserves the right to revise this documentation and to make changes in content from time to time without obligation on the part of Palm, Inc. to provide notification of such revision or changes. PALM, INC. MAKES NO REPRESENTATIONS OR WARRANTIES THAT THE DOCUMENTATION IS FREE OF ERRORS OR THAT THE DOCUMENTATION IS SUITABLE FOR YOUR USE. THE DOCUMENTATION IS PROVIDED ON AN "AS IS" BASIS. PALM, INC. MAKES NO WARRANTIES, TERMS OR CONDITIONS, EXPRESS OR IMPLIED, EITHER IN FACT OR BY OPERATION OF LAW, STATUTORY OR OTHERWISE, INCLUDING WARRANTIES, TERMS, OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND SATISFACTORY QUALITY.

TO THE FULL EXTENT ALLOWED BY LAW, PALM, INC. ALSO EXCLUDES FOR ITSELF AND ITS SUPPLIERS ANY LIABILITY, WHETHER BASED IN CONTRACT OR TORT (INCLUDING NEGLIGENCE), FOR DIRECT, INCIDENTAL, CONSEQUENTIAL, INDIRECT, SPECIAL, OR PUNITIVE DAMAGES OF ANY KIND, OR FOR LOSS OF REVENUE OR PROFITS, LOSS OF BUSINESS, LOSS OF INFORMATION OR DATA, OR OTHER FINANCIAL LOSS ARISING OUT OF OR IN CONNECTION WITH THIS DOCUMENTATION, EVEN IF PALM, INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Palm Computing, Palm OS, Graffiti, HotSync, and Palm Modem are registered trademarks, and Palm III, Palm IIIe, Palm IIIx, Palm V, Palm Vx, Palm VII, Palm, Palm Powered, More connected., Simply Palm, the Palm logo, Palm Computing platform logo, Palm III logo, Palm IIIx logo, Palm V logo, and HotSync logo are trademarks of Palm, Inc. or its subsidiaries. All other product and brand names may be trademarks or registered trademarks of their respective owners.

IF THIS DOCUMENTATION IS PROVIDED ON A COMPACT DISC, THE OTHER SOFTWARE AND DOCUMENTATION ON THE COMPACT DISC ARE SUBJECT TO THE LICENSE AGREEMENT ACCOMPANYING THE COMPACT DISC.

Palm File Format Specification
Document Number 3008-004
May 1, 2001
For the latest version of this document, visit
http://www.palmos.com/dev/tech/docs/.

Palm, Inc.
5470 Great America Pkwy.
Santa Clara, CA 95052
USA
www.palmos.com

Table of Contents

1

Introduction to File Formats

Currently, there are three types of file formats that are commonly used in the Palm OS® platform:

- Palm™ database (PDB)
- Palm query application (PQA)
- Palm resource (PRC)

Files with a `.pdb` or `.pqa` extension are record databases. Files with a `.prc` extension are resource databases. Please note, however, that the filename and extension on the desktop do not determine the name or type of database created on the handheld. The database header information inside the file determines a database name and type.

NOTE: Resource databases contain resources, not records; however, in some places the documentation and structure types use the term record generically to refer to the individual data entities stored inside of databases, including resource databases.

This book describes each of the three file formats listed above, in the following chapters:

- This chapter provides an overview of the common characteristics of all of the file formats described in this book, including the database header that is used for each format.
- Chapter 2, "PDB and PRC Database Formats," on page 21 describes the PDB and PRC file formats, which are almost identical.
- Chapter 3, "PQA Database Format," on page 29 describes the PQA file format.

- Chapter 4, "PQA Encoding Format," on page 43 describes the data encoding used in PQA files.
- Chapter 5, "PQA Tag Reference," on page 57 provides reference information for each PQA tag type used in PQA files.

About the File Format Types

This section provides an introduction to the three file format types that are described in this book. Each file format type is stored as a database.

In general, a database contains header information and a sequential list of records or resources. In addition, each database can contain one or two pieces of free-form data whose format is defined by the application that created it. The records within a database are similarly structured with record header information and record data.

File Formats Versus Memory Formats

This book describes the format of Palm databases that are stored in files on desktop computers. When one of these databases is loaded into a Palm Powered™ handheld, the database is stored in memory in a format that is similar to, but different than the format described in this book. The in-memory format of Palm databases is subject to change and is not documented by Palm, Inc.

Databases are typically imported into handheld devices when a user performs a HotSync® operation that installs an application. When a database is imported into a Palm Powered handheld, the Palm OS converts the database into standard Memory Manager objects. The Memory Manager tracks the size of each record or resource, and thus adds memory overhead; this means that the size of a database on the device is larger than its size on the desktop computer.

NOTE: The databases stored in ROM on Palm Powered handhelds are stored in a memory format, not in the file formats described in this book.

Palm Database (PDB) Files

A PDB is a record database generally used to store data for an application.

Palm Resource (PRC) Files

A Palm resource file contains a different type of data (resources instead of records), but has an almost identical structure to a PDB file. Palm OS applications are resource databases. A Palm OS application contains code resources as well as user interface resource elements.

Palm Web Clipping Application (PQA) Files

A PQA is a PDB that contains world-wide web content. On the Palm device all PQAs are associated through the Launcher with the Web Clipping Application Viewer (*Viewer*) software. When a user opens a PQA file for viewing, the Applications Launcher starts the Viewer, which in turn displays the contents of the selected PQA.

NOTE: The acronym PQA stands for "Palm Query Application." Beginning with version 4.0 of the Palm OS, these databases are referred to as "Web Clipping Applications" and the acronym WCA is used instead. The suffix for these databases remains `.pqa`.

Data Structures

The objects in Palm Database files can be represented by C structures, which are described in the chapters that follow.

About Records and Resources

Records and resources are both blocks of memory that contain any data you want. The exact definition of a record or resource is up to the application. From a low-level perspective, the difference between records and resources is the size and contents of the header for each object.

Records and resources are used for different purposes:

- Records are used to store application data such as memos or address book entries.
- Records are used to store web content in PQA databases.
- Resources are used to store the code and user interface objects for an application.

You can treat records and resources as ordered or unordered databases. You can use a callback function to sort record databases; however, you cannot sort resources on a device. You can compare two records to determine the order in which they belong; however, an index does not exist.

About Database Formats

Each database is stored in a file on a desktop computer in sequential format, as shown in Figure 1.1. The format of each database file is logically structured as shown in Figure 1.2.

Each database contains the following component parts:

- a database header that describes the database, references the `appInfo` and `sortInfo` blocks, and contains the record list, which references each record in the database
- an optional application information (`appInfo`) block in which you can store information specific to your application
- an optional sorting information (`sortInfo`) block in which you can store unique ID cross-reference tables or other meta information
- raw record or resource data

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

Figure 1.1 Database Storage Format

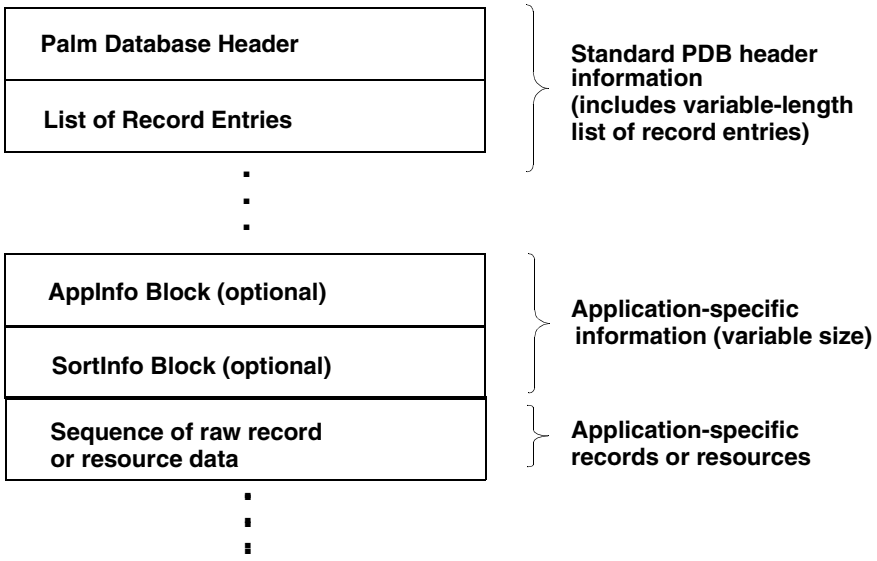
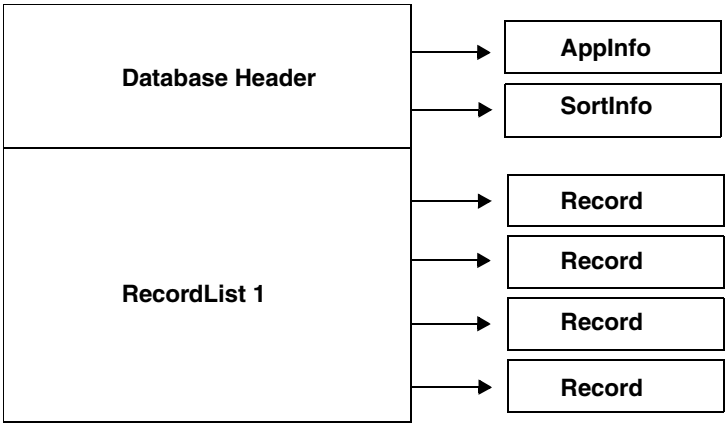


Figure 1.2 shows the logical representation of a record database file, with the header referencing the application information and sort information blocks, and with each record list referencing the raw data for the records stored in the database. The logical representation of a resource database file is the same, except that the record lists that refer to raw record data are replaced by resource lists that refer to raw resource data. The logical representation of a web clipping application database is also very similar.

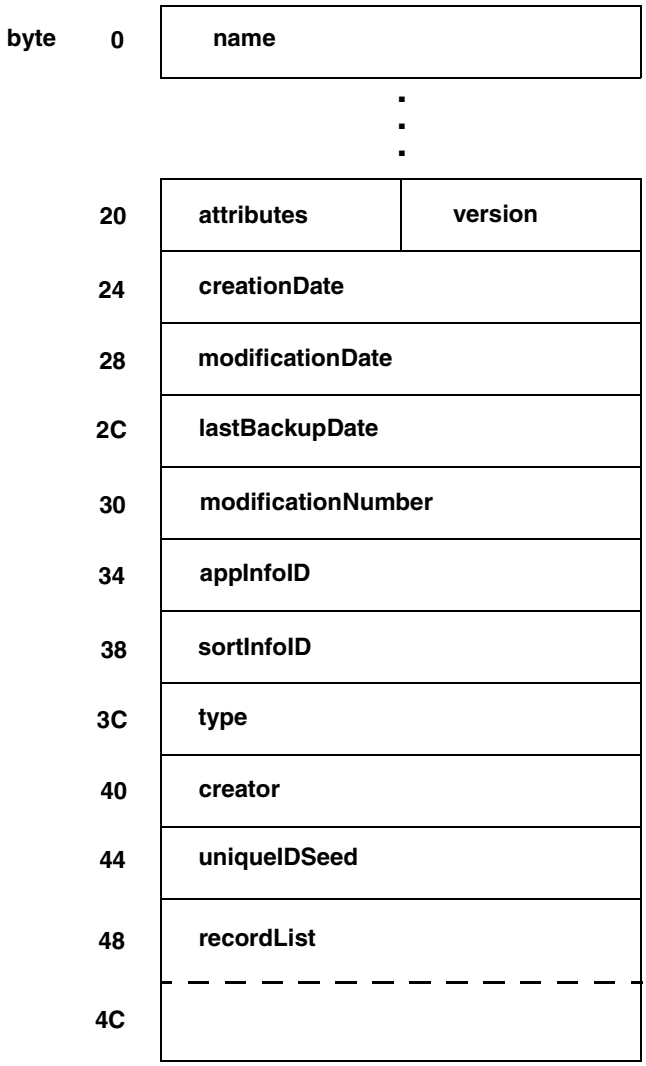
Figure 1.2 Logical Database Format for a Record Database



The Palm Database Header

The Palm database header is a standard `DatabaseHdrType` structure that is used to represent the header in PDB, PRC, and PQA database files. The format of the header is shown in Figure 1.3. The byte values shown are offsets, in hexadecimal, from the beginning of the database (and of the header).

Figure 1.3 Palm database header



Note that the structure shown in Figure 1.3 is how the header of a Palm Database is represented in a file on a desktop computer.

Palm Database Header Structure

The following structure represents a database file header:

```
typedef struct {
  UInt8           name[dmDBNameLength];
  UInt16          attributes;
  UInt32          creationDate;
  UInt32          modificationDate;
  UInt32          lastBackupDate;
  UInt32          modificationNumber;
  LocalID         appInfoID;
  LocalID         sortInfoID;
  UInt32          type;
  UInt32          creator;
  UInt32          uniqueIDSeed;
  RecordListType  recordList;
} DatabaseHdrType;
```

Field Descriptions

`name` — A 32-byte long, null-terminated string containing the name of the database on the Palm Powered handheld. The name is restricted to 31 bytes in length, plus the terminator byte.

This name is also used to create the file name of the PDB when it is backed up during the HotSync process.

`attributes` — The attribute flags for the database.

For PQA databases, this field always has the value `dmHdrAttrBackup | dmHdrAttrLaunchableData`

`version` — The application-specific version of the database layout.

`creationDate`	The creation date of the database, specified as the number of seconds since 12:00 A.M. on January 1, 1904.
`modificationDate`	The date of the most recent modification of the database, specified as the number of seconds since 12:00 A.M. on January 1, 1904.
`lastBackupDate`	The date of the most recent backup of the database, specified as the number of seconds since 12:00 A.M. on January 1, 1904.
`modificationNumber`	The modification number of the database.
`appInfoID`	The local offset from the beginning of the database header data to the start of the optional, application-specific `appInfo` block. This value is set to `NULL` for databases that do not include an `appInfo` block.
`sortInfoID`	The local offset from the beginning of the PDB header data to the start of the optional, application-specific sortInfo block. This value is set to `NULL` for databases that do not include an `sortInfo` block
`type`	The database type identifier. For PDB databases, the value of this field depends on the creator application. For PRC databases, this field usually has the value `'appl'`. For PQA databases, this field always has the value `'pqa'`.

`creator`	The database creator identifier. For PQA databases, this feld always has the value `'clpr'`.
`uniqueIDSeed`	Used internally by the Palm OS to generate unique identifiers for records on the Palm device when the database is loaded into the device. For PRC databases, this value is normally not used and is set to `0`. For PQA databases, this value is not used, and is set to `0`.
`recordList`	A list of the records or resources in the database, as described in the next section.

IMPORTANT: There is always a gap between the final record list in the header and the first block of data in the database, where the first block might be one of the following: the appInfo block, the sortInfo block, raw record or resource data, or the end of the file. The gap is traditionally two bytes long; however, if you write code to parse a database, your code should be able to handle any size gap, from zero bytes long and up.

The Record List

The Palm database header ends with a record list. The record list has its own header, followed by 0 or more record entries. Each record entry describes a single record in the file.

The record list has a variable length. When the database is loaded into a Palm Powered handheld, the Palm OS attempts to grow the list. If it cannot grow the list, the OS creates another record list and links it to the previous one by filling in the `nextRecordListID` field with the location of the new list. This capability is rarely used,

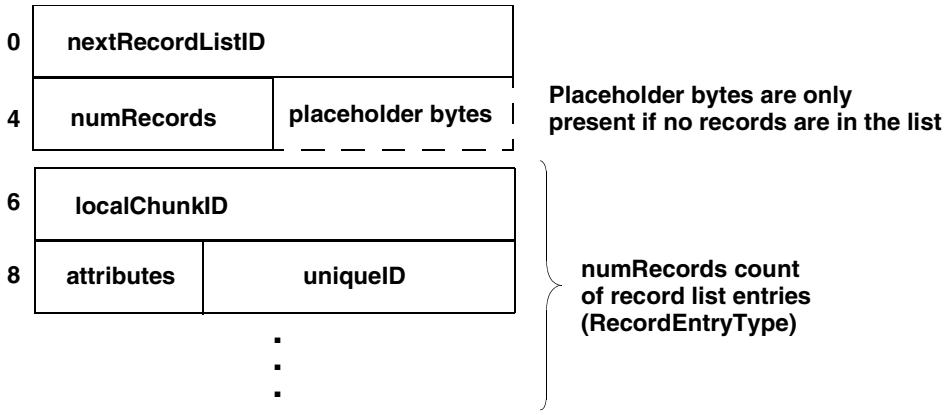
and its use is discouraged by Palm. For more information, see "About Multiple Record or Resource Lists in a Database" on page 17.

Each record entry references the location of the raw data for the record or resource and contains the attribute and ID information for that record or resource.

The remainder of this chapter describes the record list structure. However, the format of the record entries is different for different Palm database types. The record entry format for PDB databases and the resource entry format for PRC databases are shown in Chapter 2, "PDB and PRC Database Formats." The record layout format for PQA databases is shown in Chapter 3, "PQA Database Format."

Figure 1.4 shows the structure of a record list.

Figure 1.4 Palm Database record list



Palm Database Record List Structure

The following structure declaration represents a Palm Database record list:

```
typedef struct {
  LocalID nextRecordListID;
  UInt16  numRecords;
  UInt16  firstEntry;
} RecordListType;
```

NOTE: The placeholder bytes shown in Figure 1.4 appear at the end of the record list, if there is one. If there is no list, these bytes appear just after the list header; otherwise, they appear after the last entry in the list.

Field Descriptions

`nextRecordListID`	The local chunk ID of the next record list in this database. This is `0` if there is no next record list, which is almost always the case. For more information, see "About Multiple Record or Resource Lists in a Database" on page 17.
`numRecords`	The number of record entries in this list.
`firstEntry`	The start of an array of record entry structures, each of which represents a single record in the list.

About Multiple Record or Resource Lists in a Database

The structure of Palm databases allows for multiple record lists in a single database; the record lists are chained together by setting the `nextRecordListID` field of the first record list to the offset of the next list in the database.

In practice, this capability is very rarely used, and the `nextRecordListID` field in the database header is almost always set to 0, which indicates that there is only one record list in the database. Since a single record list can be used to describe the maximum number of records (64K) in a file, multiple record lists are never required.

Palm, Inc. recommends against building databases with chained headers, and that your parsing code reject databases that have a

non-zero value in the `nextRecordListID` field, to avoid potentially truncating such a database if your code encounters one.

A database with chained record lists might be encountered under very specific circumstances:

- when a huge database (one containing more than approximately 6000 records that has caused the headers to fragment) is beamed to a desktop OBEX stack from a Palm handheld device running version 3.5 or earlier of the Palm OS
- when code on a Palm handheld device uses the `ExgDbWrite` function to produce a PRB or PRC file image from such a database

NOTE: Version 4.0 and later of the Palm OS never produces chained record lists.

The Application and Sort Information Blocks

The database header can reference two optional application-specific blocks of information:

- The sort information (`sortInfo`) block
- The application information (`appInfo`) block

The `sortInfo` block is under your control. The OS does not use `sortInfo`. You can use it to store meta information about the database.

You are free to include whatever data you want in the `appInfo` block. However, there are restrictions on how you use this block if one of the following applies:

- your application uses Palm OS category functionality, as described in Chapter 2, "PDB and PRC Database Formats," on page 21.
- the database has the `dmHdrAttrLaunchableData` attribute, as described in Chapter 3, "PQA Database Format," on page 29.

About Third Party Tools

There are a number of third party tools available for creating Palm databases on desktop computers, and for converting images in various formats into Palm image format. Rather than include a partial list, Palm, Inc. encourages you to search on the Internet for these tools, and recommends the following search terms:

- convert pdb
- convert pqa
- convert prc

Additional Resources

- Documentation

  Palm publishes its latest versions of this and other documents for Palm OS developers at

  http://www.palmos.com/dev/tech/docs/

- Training

  Palm and its partners host training classes for Palm OS developers. For topics and schedules, check

  http://www.palmos.com/dev/tech/support/classes/

- Knowledge Base

  The Knowledge Base is a fast, web-based database of technical information. Search for frequently asked questions (FAQs), sample code, white papers, and the development documentation at

  http://www.palmos.com/dev/kb/

2

PDB and PRC Database Formats

This chapter describes the format of Palm OS® record (PDB) and resource (PRC) databases. Palm™ record databases contain records that are used with applications that run on Palm Powered™ handhelds. Palm resource databases contain application resources, including the code and the user interface objects for the application.

These databases are stored in memory on handheld devices, and are stored in file form on desktop computers. This chapter describes the file format of these databases, which is slightly different than their in-memory format. The in-memory format is subject to change and is not documented by Palm, Inc.

This chapter contains the following sections:

For an overview of Palm databases and file formats, including a detailed description of the database header format, see Chapter 1, "Introduction to File Formats."

NOTE: This chapter describes the format of PDB and PRC databases that are stored in files on desktop computers. When one of these databases is loaded into a Palm Powered handheld, the database is stored in memory in a format that is similar to, but different from the format described here. The in-memory format of PDBs and PRCs is subject to change and is not documented by Palm, Inc.

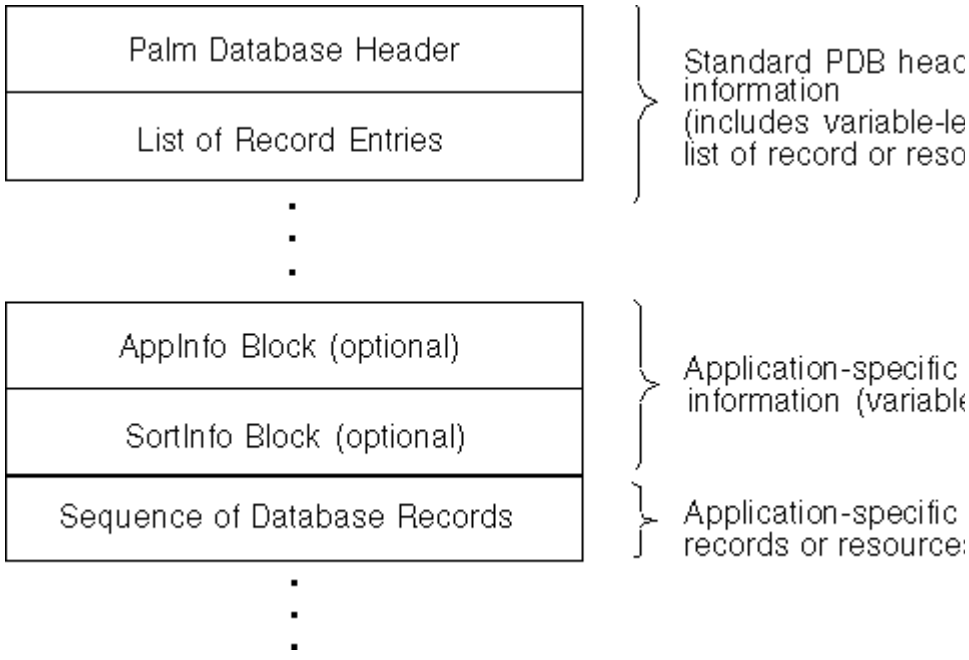
Overview of PDB and PRC Databases

Each PDB and PRC database contains the following components:

- A header containing fields that describe the database and refer to the information blocks and raw record data in the database. The Palm Database header is described in "The Palm Database Header" on page 12.
- A list of record entries, each of which describes a block of raw record or resource data.
- Two optional information blocks: the application information block and the sort information block.
- The raw record data, which is stored in linear format and referenced from the record list in the header.

Figure 2.1 shows the structure of a Palm database, as stored in a file on a desktop computer.

Figure 2.1 PDB and PRC database format



Record and Resource Entries

The record list in the Palm database header contains a list of entries that describe the raw data records or resources in the database. The record list is described in "The Record List" on page 15. The entries in PDB and PRC databases have different structures, and are described separately in this section.

PDB Record Entries

The following structure declaration represents a record entry in a PDB file:

```
typedef struct {
  LocalID localChunkID;
  UInt8   attributes;
  UInt8   uniqueID[3];
} RecordEntryType;
```

Field Descriptions

`localChunkID`	The local offset from the top of the PDB to the start of the raw record data for this entry. Note that you can determine the size of each chunk of raw record data by subtracting the starting offset of the chunk from the starting offset of the following chunk. If the chunk is the last chunk, it's end is determined by the end of the file.
`attributes`	Attributes of the record.
`uniqueID`	A three-byte long unique ID for the record.

PRC Resource Entry Fields

The following structure declaration represents a resource entry in a PRC file:

```
typedef struct {
  UInt32    type;
  UInt16    id;
  LocalID   localChunkID;
} RsrcEntryType;
```

Field Descriptions

`type`	The resource type.
`id`	The ID of the resource.
`localChunkID`	The local offset from the top of the PRC to the start of the resource data for this entry. Note that you can determine the size of each chunk of raw resource data by subtracting the starting offset of the chunk from the starting offset of the following chunk. If the chunk is the last chunk, it's end is determined by the end of the file.

The Application Information Block

Each Palm Database can optionally include an application information (appInfo) block that contains arbitrary information.

The format of the `appInfo` block is determined by the creator of the database. However, PDBs that support the standard Palm OS category data, the `appInfo` block contains specific information, as described in "Finding the Length of the Application Information Block," below.

NOTE: PRCs can also contain application information blocks; however, this is rarely the case.

Finding the Length of the Application Information Block

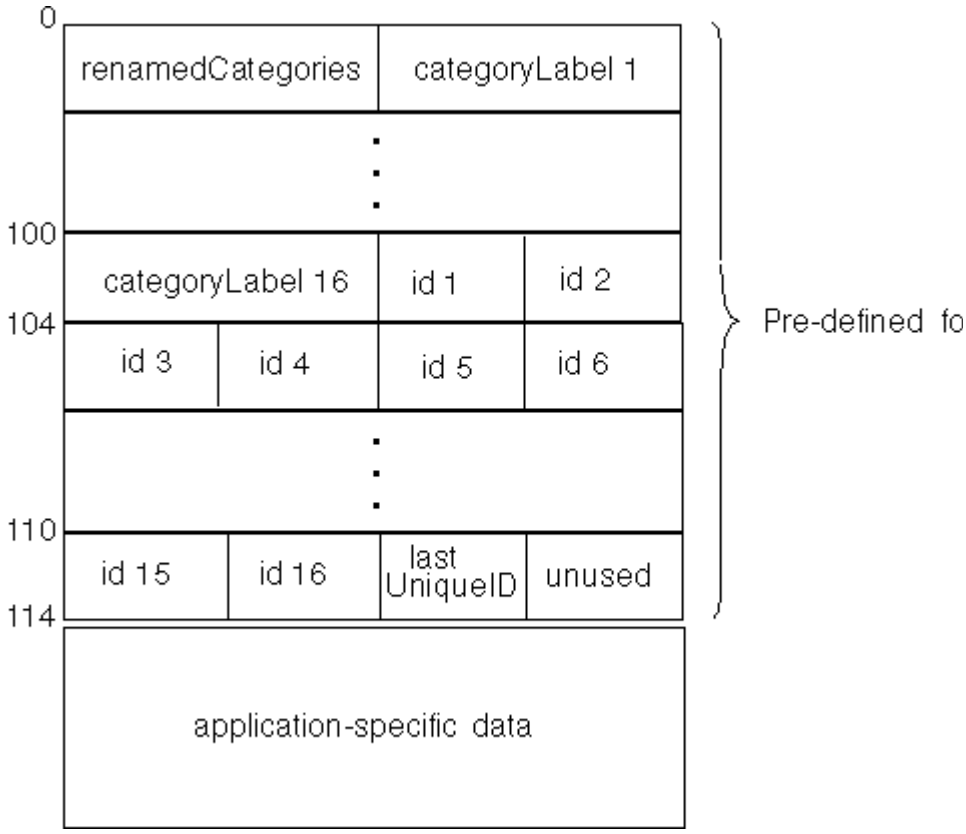
If the database includes an application information block, you can find its length by finding the block that follows it:

- If the database includes a sort information block, that block immediately follows the application information block.
- If the database does not include a sort information block, but does include one or more records, then the end of the application information block is just before the start of the first record block.
- If the database does not contain a sort information block and does not contain any records, then the end of the application information block is the end of the file.

Standard Category Data in an Application Information Block

A PDB for an application that supports standard Palm OS category data includes the category data in the standard format shown in Figure 2.2.

Figure 2.2 PDB appInfo bock for standard category data



The following structure declaration represents an application information block for an application that uses standard Palm OS category information:

```
typedef struct {
  UInt16  renamedCategories;
  Char    categoryLabels[16][16];
  UInt8   categoryUniqIDs[16];
  UInt8   lastUniqID;
  UInt8   padding;
} AppInfoType;
```

Field Descriptions

`renamedCategories`	Specifies which categories have been renamed.
`categoryLabel`	An array of 16 null-terminated category labels, each of which is 16 bytes long.
`categoryID`	An array of 16 category ID values, each of which is one byte long.
`lastUniqID`	The last unique category ID assigned.
`padding`	Unused.

The Sort Information Block

The structure of the optional `sortInfo` block is completely up to the application; there is no standard format, nor is there a structure declared for this block. Most PDBs that contain a `sortInfo` block use it to store ordering information based on record IDs.

PRCs can contain a `sortInfo` block, but they rarely do.

Finding the Length of the Sort Information Block

If the database includes a sort information block, you can find its length by finding the block that follows it:

- If the database includes one or more records, then the end of the sort information block is just before the start of the first record block.
- If the database does not contain any records, then the end of the sort information block is the end of the file.

PDB and PRC Raw Data

Record data in a PDB is stored as a block of contiguous records. The local offset to the beginning of each record is stored in the record list(s) in the database header. The length and format of the record data is application-specific.

Similarly, resource data in a PRC database is stored as a block of contiguous resources. The local offset to the beginning of each resource is stored in the record list(s) in the database header. The length and format of the resource data is not documented in this book.

Reading and Writing PDB and PRC Data

The Palm OS provides functions that you can use to convert data into or out of Palm Database formats:

- If you have a chunk of data on the handheld device that is formatted as described in this chapter, you can use either the `DmCreateDatabaseFromImage` function or the `ExgDBRead` function to convert that data into a Palm Database.
- If you want to convert a Palm Database into a chunk of data on the handheld device, you can use the `ExgDbWrite` function.

The `DmCreateDatabaseFromImage`, `ExgDBRead`, and `ExgDbWrite` functions are documented in the *Palm OS Programmer's API Reference*.

3

PQA Database Format

Palm web clipping applications contain HTML content that can be displayed on Palm Powered™ handhelds; typically each such application contains an HTML form with which the user can interact to access information on the Internet.

Each web clipping application (*WCA*) is a database that is opened by the Web Clipping Application Viewer (*Viewer*) program: when the user taps on a WCA in the Palm OS® Application Launcher, the Launcher launcher the Viewer, which displays the home page of the web clipping application.

Web clipping applications were originally called Palm Query Applications, the Viewer was originally called the Clipper, and each web clipping application was stored as a `.pqa` file. Starting with version 4.0 of the Palm OS, the applications are called web clipping applications; however the `.pqa` suffix is still used, and `'clpr'` is still used as the creator ID for web clipping applications.

This chapter uses the term **PQA database** to refer to the format of web clipping applications as stored in files on desktop computers. Note that a PQA database is a PDB database with records that contain web content in a specific format, and thus the logical structure of a PQA file is the same as the logical structure of a PDB file.

This chapter contains the following sections:

- "PQA Overview" provides an overview of the PQA representation and shows an image of that representation.
- "PQA Application Information Block" describes the application information block in a PQA, which contains information about the web clipping application.
- "Web Content Records" describes the web content records that are stored in PQA databases.

NOTE: This chapter describes the format of PQA databases that are stored in files on desktop computers. When one of these databases is loaded into a Palm Powered handheld, the PQA is stored in memory in a format that is similar to, but different than the format described here. The in-memory format of PQAs is subject to change and is not documented by Palm, Inc.

For more information about web clipping applications, see *Web Clipping Developer's Guide*.

PQA Overview

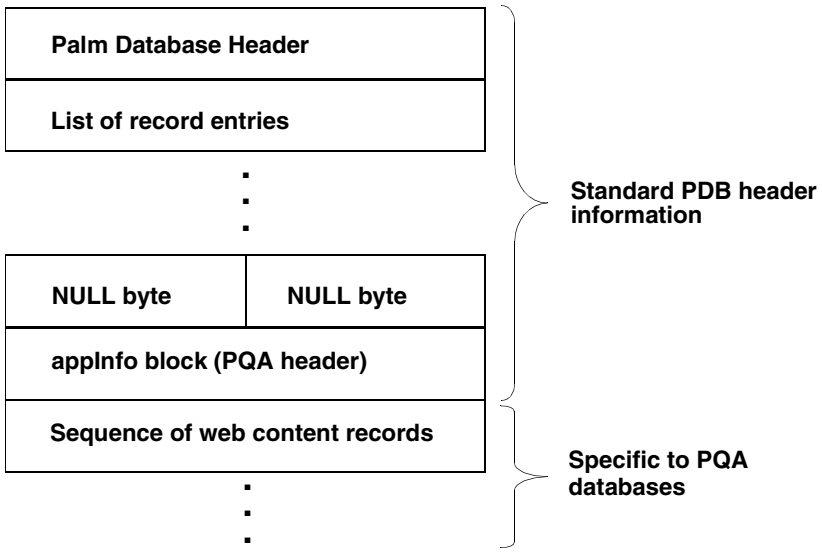
Each PQA database contains the following components:

- A header containing fields that describe the database and refer to the application information block and web content records in the database. The PQA header has the same structure as other Palm databases, and is described in "The Palm Database Header" on page 12.
- The application information block, which contains specific PQA information and is described in "PQA Application Information Block" on page 33.
- The web content record data, which is stored in linear format and referenced from the record list in the header. For more information, see "Web Content Records" on page 36.

NOTE: PQA databases never contain the optional `sortInfo` block that can be stored in Palm databases.

Figure 3.1 shows the structure of a PQA database, as stored in a file on a desktop computer.

Figure 3.1 PQA database format



How PQAs are Different Than PDBs and PRCs

PQA database files have the same structure as do PDB and PRC databases. However, the information stored in PQA database files has some specific differences:

- The `appInfo` block is always present in a PQA database and always contains specific information, as described in "PQA Application Information Block" on page 33
- The `sortInfo` block is never present in a PQA database.
- The record list entries in PQA databases have the same structure as the entries in PDB database, but some fields are not used, as described in "PQA Record Entries" on page 32.
- The raw record data in PQA databases has a specific structure, as described in "Web Content Records" on page 36.
- Certain fields in the PQA database header contain specific values:
  - the `attributes` field always contains the value `dmHdrAttrBackup | dmHdrAttrLaunchableData`
  - the `type` field always contains the value `'pqa'`
  - the `creator` field always contains the value `'clpr'`

PQA Record Entries

The record list in the PQA header contains a list of PQA record entries. Each entry describes a web content record that is stored in the database.

The record list in PQA databases is the same as the record list in other Palm databases. For more information, see "The Record List" on page 15.

The following structure declaration represents a record entry in a PQA file. Note that this is the same structure that is used in PDB database files; however, two of the fields in the structure are always set to 0 in PQA files.

```
typedef struct {
  LocalID localChunkID;
  UInt8   attributes;
  UInt8   uniqueID[3];
} RecordEntryType;
```

Field Descriptions

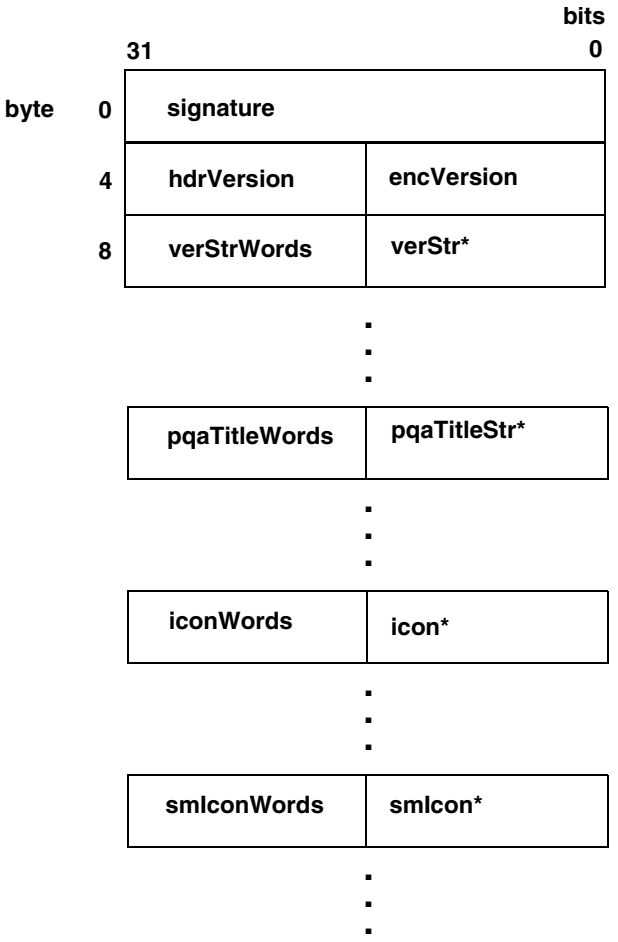
`localChunkID`	The local offset from the top of the PQA to the start of the web content record's header for this entry. See "Web Content Records" on page 36 for more information about the format of the records. Note that you can determine the size of each chunk of raw record data by subtracting the starting offset of the chunk from the starting offset of the following chunk. If the chunk is the last chunk, it's end is determined by the end of the file.
`attributes`	Always set to `0` in PQA databases.
`uniqueID`	Always set to `0` in PQA databases.

PQA Application Information Block

The header in each PQA file refers to an application information block that provides specific information about the web clipping application. The structure of this block is shown in Figure 3.2.

NOTE: Field names that are shown ending with an asterisk (*) in Figure 3.2 are variable-length fields that are padded, if necessary, to the next word boundary.

Figure 3.2 PQA appInfo Block



Field Descriptions

`signature`	This is always set to `'lnch'` (`0x6C6E6368`).
`hdrVersion`	The version number for this PQA information block.
`encVersion`	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 assumed to have the same encoding version.
`verStrWords`	The length of the string in the `verStr` field, specified as the number of 16-bit words, including any pad byte at the end.
`verStr`	A sequence of (`verStrWords` * 2) bytes. This is a null-terminated version string that the Viewer displays, and represents the version information for the web clipping application. If the value of `verStrWords` is zero, this field contains zero 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`	The length of the string in the `pqaTitleStr` field, specified as the number of 16-bit words, including any pad byte at the end.

`pqaTitleStr`	A sequence of (`pqaTitleWords` * 2) bytes. This is a null-terminated version string that the Launcher displays for this PQA's icon, and represents the title string for the PQA itself. This is not the title string included in the original home page HTML source code, which is shown when the Viewer displays that page. If the value of `pqaTitleWords` is zero, this field contains zero 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`	The length of the bitmap data in the `icon` field, specified as the number of 16-bit words, including any pad byte at the end.
`icon`	A sequence of (`iconWords` * 2) bytes. This is a Palm bitmap (`BitmapType` and associated bitmap data) that represents the large icon that appears on the device for this PQA.[a] If the value of `iconWords` is zero, this field contains zero 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.

`smIconWords`	The length of the bitmap data in the `smIcon` field, specified as the number of 16-bit words, including any pad byte at the end.
`smIcon`	A sequence of (`smIconWords` * 2) bytes. This is a Palm bitmap (`BitmapType` and associated bitmap data) that represents the small icon that appears on the device for this PQA.[a] If the value of `smIconWords` is zero, this field contains zero 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.

a. The icon sizes are 32 by 32 for the large icon and 15 by 9 for the small icon. There is no color table present in these bitmaps. Currently, images converted to Palm bitmaps for use as icons have their color depth reduced to 1 bit per pixel.

Web Content Records

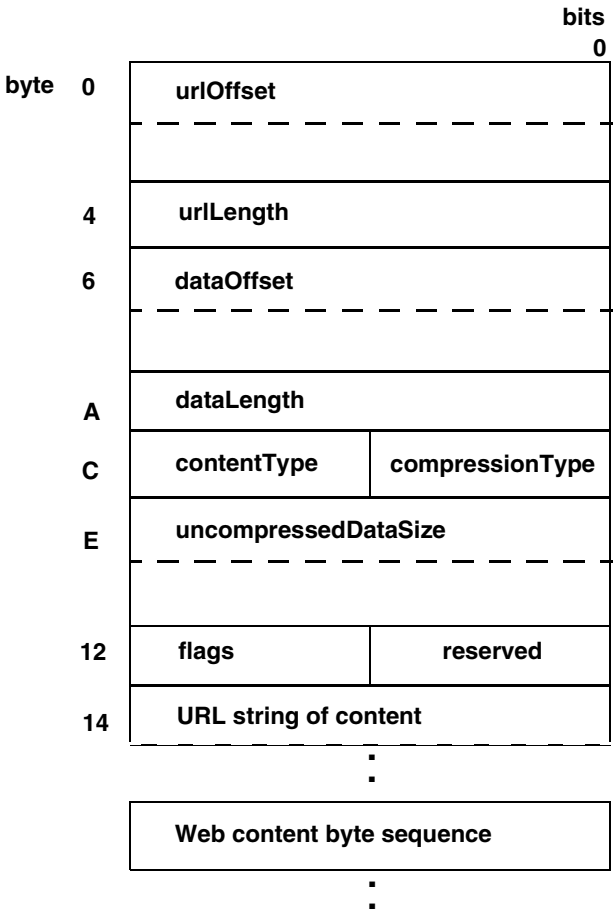
Following the `appInfo` block in a PQA file is a sequence of web content records; one for each record list entry in the PDB header record list.

Each web content record begins on a word boundary, and contains:

- A record header.
- The content's original URL.
- The content itself, which is either HTML data encoded into PQA format or graphic data.

The layout of a web content record is shown in Figure 3.3, and the fields in the records are described below.

Figure 3.3 Web Content Record



Field Descriptions

`urlOffset`	The offset, in bytes, from the top of this `record` 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 field is included for historical reasons.
`urlLength`	The length of the URL string, in bytes. This is the size of the URL string itself, without including the null terminator byte or any pad byte that follows the string data.
`dataOffset`	The offset, in bytes, from the top of this record header to the start of the data for this web content resource.
`dataLength`	The length of the data, in bytes.
`contentType`	A code for the type of content, defined in `CMLConst.h` and described in "Web Content Record Content Types" on page 40. 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.
`compressionType`	A code for the type of compression used for the content, defined in `CMLConst.h` and described in "Web Record Compression Types" on page 41.

`uncompressedDataSize`	The uncompressed size, in bytes, of the web content. If the compression type of the record is `cmlCompressionTypeNone`, the value of this field equals the value of the `dataLength` field. If the web content is an image, this field contains the size of the Palm OS bitmap data before compression.
`flags`	Unused and set to `0`.
`reserved`	Currently unused and set to `0`.
`URL string`	The URL string, which follows the end of the header. This 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, 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. The data can contain various content types, as explained in the `contentType` field description. For more information about the format used to encode data, see and Chapter 4, "PQA Encoding Format."

Web Content Record Content Types

The content type constants are used in the `contentType` field of each content record to specify the type of data found in a web content record in a PQA database.

Constant	Value	Description
`cmlContentTypeTextPlain`	`0`	Plain text
`cmlContentTypeTextHTML`	`1`	HTML text
`cmlContentTypeImageGIF`	`2`	GIF images
`cmlContentTypeImageJPEG`	`3`	JPEG images
`cmlContentTypeTextCml`	`4`	CML text
`cmlContentTypeImagePalmOS`	`5`	Palm OS bitmap image format

The current version of the Viewer processes only content identified with either content type `cmlContentTypeTextCml` or `cmlContentTypeImagePalmOS`.

The web content encoder, which is used by both the WCA Builder application and the Palm proxy server, interprets the following resources as "plain text" content:

- MIME type text/plain
- any text that is not identified as text/html, image/gif, or image/jpeg

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 Viewer handles.

Content type `cmlContentTypeImagePalmOS` is standard Palm OS bitmap image data, which may be compressed according to the Palm OS bitmap standard. The encoder converts MIME content types image/gif and image/jpeg into compressed Palm OS bitmaps.

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

The encoder converts resources with MIME type text/html to content with type `cmlContentTypeTextCml`. The format of the data is specified in Chapter 4, "PQA Encoding Format."

Web Record Compression Types

The compression type constants specify the type of compression applied to the data found in a web content record in a PQA database.

Constant	Value	Description
`cmlCompressionTypeNone`	`0`	Uncompressed data
`cmlCompressionTypeBitPacked`	`1`	Data compressed in Palm bit-packed format, which is described in Chapter 4, "PQA Encoding Format," on page 43.
`cmlCompressionTypeLZ77`	`2`	Data compressed in Lz77 format. The Palm implementation of Lz77 format is described in an appendix in the *Web Clipping Developer's Guide*.

Type `cmlCompressionTypeNone` is an intermediate form; the WCA Builder and Palm proxy server always generate `cmlCompressionTypeBitPacked` data. For more information, see the sections "About Bit Packed Compression" on page 46 and "Unpacked Notation" on page 52.

4

PQA Encoding Format

This chapter describes the web clipping application (*PQA*) data encoding format, which is a compressed data format shared by the web clipping application viewer (*Viewer*) program, the Palm proxy servers, and the web clipping application builder (*WCA Builder*) program.

The PQA encoding format is a binary translation of HTML source data known as Compressed Markup Language (*CML*). This data is compressed with a bit-packed scheme that is proprietary to Palm, Inc.

This chapter describes the PQA data encoding format and the associated bit-packed compression scheme, in the following sections:

- "About PQA Data" provides an overview of the PQA data format and how it differs from standard HTML.
- "The PQA Data Format" on page 46 describes specific elements of the PQA data format.
- "Unpacked Notation" on page 52 describes the unpacked, intermediate representation of PQA data streams, which you can use when debugging a PQA encoder.

This document describes the version of the PQA encoding and compressing scheme that is current with release 4.0 of the Palm OS®.

About PQA Data

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 newline characters; other PQA tags and their parameters are embedded as

binary data rather than ASCII characters, as described in the section "Unpacked Notation" on page 52).

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 file. 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.

The PQA encoding form results in a compact representation relative to the size of the original HTML, as shown in the next section.

An Example of Converting HTML to PQA Format

This section provides an example of translating HTML input into bit-packed PQA format. Listing 4.1 shows the original HTML file.

Listing 4.1 Sample HTML file before conversion to PQA

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

The unpacked PQA data representation of the file is shown in Listing 4.2, the top line of which lists the hexadecimal value of each byte, and the bottom line of which lists the corresponding ASCII data.

Listing 4.2 The unpacked PQA representation of the HTML file

```
45 78 61 6D 70 6C 65 00 42 6F 64 79 20 74 65 78 74 01 71
E  x  a  m  p  l  e \0  B  o  d  y sp  t  e  x  t cmlEnd
```

Listing 4.3 shows the bit-packed data representation of the PQA data, in hexadecimal.

Listing 4.3 The PQA data in bit-packed format

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

Note that the hexadecimal representation above includes zero bits that are not actually part of the PQA bit stream, which actually ends with the last "on" (1) bit in the byte with the value 40h.

For an explanation of the bit-packed representation used in PQA files, see "About Bit Packed Compression" on page 46.

The remainder of this chapter describes how HTML elements are encoded and compressed into PQA format to produce a bit stream like the one shown in this section.

How PQA Differs From HTML

The major emphasis of the PQA format is that it is optimized for size. This was done to promote speedier transmission of data using wireless communications, which are currently slower than typical dial-up connections from home computers. The PQA format sacrifices some readability and flexibility in exchange for enhanced compactness.

One major design difference between HTML and PQA format is that PQA format is not designed as a content creation language. It is instead 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 newline 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 PQA 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 the HTML markup appendix of the *Web Clipping Developer's Guide*.

The PQA Data Format

This section describes the PQA data format, in the following sections:

- "About Bit Packed Compression" describes the bit-packed compression scheme used in the PQA data format.
- "Representing Text in PQA Format" on page 49 describes how text is represented in PQA data format.
- "Data Termination" on page 52 describes the tag used to terminate a PQA data stream.

Note that the effects of a tag in the PQA data stream are ended either by a `cmlCharEndCharacter`, or by the appearance in the data stream of another tag that overrides the previous effects.

About Bit Packed Compression

In its raw form, unpacked PQA data is an encoded form of HTML, smaller in size than the original content, and is considered to have the compression type `cmlCompressionTypeNone`.

The WCA Builder program, the Palm proxy servers, and the Viewer program all work with PQA data that has been compressed using a proprietary, bit-packed compression scheme. This data is considered to have the `cmlCompressionTypeBitPacked` compression type.

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, as noted below, appears in the stream, each sequence of 5 bits is assumed to represent a single text character. Table 4.1 lists the possible 5-bit characters.

Table 4.1 Bit-Packed Encoding 5-bit Characters

Value	Special?	Reset ?	CML Tag	Description
0	Yes	Yes	`cmlCharEnd`	Used to end a `TextZ` type and certain tags.
1	Yes	Yes	`cmlCharStart`	Followed by an 8-bit Tag ID
2	Yes	No	`cmlCharEsc`	Single character escape; followed by a single ASCII character
3	No	No	`cmlCharFormFeed`	ASCII Formfeed (0x0c)
4	No	No	`cmlCharLineBreak`	ASCII Carriage return (0x0d)
5	No	No	`cmlCharSpace`	ASCII Space (0x20)
6-31	No	No	N/A	ASCII lowercase letters (0x61 through 0x7a)

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. A PQA encoder may produce sections of output containing 8-bit characters; however, even within these sections, the character values 0, 1, and 2 always have special meaning. For more information about using 8-bit characters, see "cmlTag8BitEncoding" on page 62.
- **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 this character.

Bit Packed Compression Encoding

As you can see, the bit-packing compression scheme saves space when applied to input consisting of lowercase ASCII text characters and HTML tags and attribute values (including image data).

Table 4.2 provides a summary of how the different data types are represented in the bit-packed compression scheme.

Table 4.2 Bit-packed Compression Encoding Summary

Encoding type	Description
ASCII text	Lowercase ASCII text characters, the space character and the newline 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.
HTML tag	Each tag is encoded as a 5-bit start tag code followed by an 8-bit tag identifier. If the tag includes attributes, then the encoding also includes encoded tag parameters, using numeric parameter values and ASCII text encoding. If the tag encloses associated tag data, such as a hyperlink tag enclosing a link or an image tag specifying an image URL, the encoding also includes encoded tag data, using ASCII text encoding and image compression. If the tag requires an end tag, such as a hyperlink tag's </A>, the encoding includes a 5-bit "end tag" code.
Numeric parameter value	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 "Data Types in PQA Format" on page 58.
Image compression	The encoder converts 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.

NOTE: The 5-bit compressor used by the WCA Builder and the Palm 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 it generates bit-packed plain text only from plain ASCII text data or as part of PQA data.

Data Parsing Modes

If you are parsing a PQA data stream, you need to be able to operate 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 specified Chapter 5, "PQA Tag Reference".

Representing Text in PQA Format

This section shows two examples of how a simple section of text is represented in PQA format. The first example is:

```
abc d
ef
```

is 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
```

If the data stream includes an 8-bit ASCII character, that character is preceded in the data stream by the single-character escape code, which has the value 2. For example, the following text contains the 8-bit uppercase character ‘C’ and thus includes the escape code.

```
a Cow
```

The above text is 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-bit or 16-bit character text string. Tags are described in the next sections.

PQA Tags

Each PQA tag in the data stream is preceded by the PQA tag start character, which is a 5-bit character with the value `1`. The tag start character is always followed by an 8-bit tag ID. And some of the tags are followed in the data stream by parameter values.

IMPORTANT: Whenever the tag start character is encountered in the data stream, the text encoding mode is reset to 5-bit character mode.

This section provides an overview of the PQA tags. For reference information on tag, see Chapter 5, “PQA Tag Reference.”

Text Encoding Tags

The PQA compression format is size-optimized for lowercase ASCII characters, each of which can be represented in 5 bits. When the data stream includes characters other than lowercase ASCII character, the data stream includes a text encoding tag, followed by those characters.

All data following the text encoding tag is assumed to be encoded in accordance with the tag, until one of the text mode reset tags (`cmlCharEnd` or `cmlCharStart`) is encountered.

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 `00000000` and the `cmlCharStart` character becomes the 8 bit sequence `00000001`.

Whenever the reset character is encountered in the data stream, the text mode reverts to 5-bit characters.

Listing 4.4 shows an example of using the `cmlTag8BitEncoding` tag to represent a a sequence of uppercase characters in the text string “a BIG dog.”

Listing 4.4 Example of using the `cmlTag8BitEncoding tag`

```
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, 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.

Added Overhead for Text Encoding Tags

Including the `cmlCharStart`, `cmlTag8BitEncoding`, and `cmlCharEnd` characters adds a fixed amount of data to the stream. This makes sense for long runs of characters that use the encoding, but does not make sense for small character runs. Instead, for small runs, you can use the single character escape code in front of each 8-bit character in the stream.

The WCA Builder program and the Palm Web Clipping Proxy servers use the following rule when encoding 8-bit text runs: if the text run includes a sequence of four or less 8-bit characters, encode each as a single character escape. If the text run includes more than four 8-bit characters, the entire text run is encoded as an 8-bit encoding, with start, stop, and encoding tags.

Data Termination

A PQA format data stream is terminated with a `cmlTagCMLEnd` tag. PQA data streams always end with the last 1 bit of the `cmlTagCMLEnd` tag value; they are not padded with any following bits.

Unpacked Notation

Originally, the PQA format was envisioned as a tag-encoding method with one representation, which 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 WCA 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.

The reason to know about the `cmlCompressionTypeNone` format is that it makes debugging a PQA data stream much easier. If you are writing a PQA encoder, you will probably want to debug using the intermediate `cmlCompressionTypeNone` data.

NOTE: The reference sections in Chapter 5 denote 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 to identify data elements of `cmlCompressionTypeNone`.

Translation of Bit-Packed to Uncompressed Data

Unpacked PQA data includes just two special characters, as shown in Table 4.3.

Table 4.3 Unpacked Encoding Characters

Value	Special	Reset	Description
0	Yes	Yes	`cmlCharEnd` character. Used to end `TextZ` data and certain tags.
1	Yes	Yes	`cmlCharStart` character, followed by an 8-bit Tag ID.

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:

Bit-packed Data	Uncompressed Data
Bit = 1	Byte = 0x01
Bit[3] = 1, 0, 1	Byte = 0x05
TextZ = “foo”	“foo”, `NULL` terminated ASCII string
Byte = 0xCD	Byte = 0xCD
IntV = -1	DWord = 0xFFFFFFFF (-1)
UIntV = 7	DWord = 0x00000007
Uint8V = 2	Byte = 0x02

Five-bit tags are treated in the following manner:

Bit-packed Data	Uncompressed Data
`cmlCharEnd` (0)	Byte = 0x00
`cmlCharStart` (1)	Byte = 0x01
`cmlCharEsc` (2)	Unused
`cmlCharFormFeed` (3)	Byte = 0x0C
`cmlCharLineBreak` (4)	Byte = 0x0D
`cmlCharSpace` (5)	Byte = 0x20
`cmlTag8BitEncoding`	Unused
`cmlCharA` (6)… `cmlCharZ` (31)	Byte = 0x61… 0x7a

You 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 bit-packed representation is shown in Listing 4.5.

Listing 4.5 Bit-packed representation of an HTML page

```
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
```

Listing 4.6 shows the same HTML code in unpacked representation.

Listing 4.6 Unpacked representation of an HTML page

```
45 78 61 6D 70 6C 65 00 42 6F 64 79 20 74 65 78 74 01 71
E  x  a  m  p  l  e \0  B  o  d  y sp  t  e  x  t cmlEnd
```

5

PQA Tag Reference

This chapter provides reference information for the tags found in PQA data streams. This chapter contains three sections:

- "Specifying PQA Data in Compact Notation" describes the notation used to specify PQA data streams.
- "PQA Tag Definitions" on page 61 presents a reference description of each PQA tag. The tags are presented in alphabetical order.
- "Summary of CML Tags" on page 107 provides a summary table that organizes the tags according to usage.

Specifying PQA Data in Compact Notation

This section describes a notation that is used in the remainder of this chapter for representing PQA data. The notation is described in the following sections:

- "About Compact Data Structure Notation" describes the notation.
- "Data Types in PQA Format" on page 58 describes the data types used in the compact data structure notation.

About Compact Data Structure Notation

This notation, known as Compact Data Structure Notation (*CDSN*), describes data elements that use the `cmlCompressionTypeBitPacked` compression. CDSN has the general form:

```
<data type> <identifier> = <legal value>
```

For example, the notation for a three bit value:

```
Bit[3] aValue = 7
```

Note that `<legal value>` may be an identifier, the value of which is a legal value. Also, note that the values of `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.).

The following is another example:

```
Bit enabled = 1
Bit[3] type = typeRound
Int16 length = 0x1234
```

The above structure represents the following sequence of 20 bits:

```
1 010 0001001000110100
```

Which breaks down as follows:

- 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 with the most-significant-bit first.

Data Types in PQA Format

This section describes the data types used in PQA format:

- Primitive Data Types
- Variable Length Integer Data Types
- Text Data Types

Primitive Data Types

A number of primitive data types are used in CDSN. The basic types are shown in Table 5.1.

Table 5.1 CDSN primitive data types

Type	Description
`Bit`	A single bit
`UInt8`	8-bit unsigned integer value
`Int8`	8-bit signed integer value
`UInt16`	16-bit unsigned integer value
`Int16`	16-bit signed integer value
`UInt32`	32-bit unsigned integer value
`Int32`	32-bit signed integer value

Variable Length Integer Data Types

CDSN also provides a number of variable length integer types, each of which uses a varying number of bits to represent different value ranges. These variable-length integer types work by using the first 1 to 4 bits to identify the number of value bits that follow.

Table 5.2 shows the number of bits used for each value range in one variable integer type: the `UIntV` data type.

Table 5.2 Total bits used for each value range for the UIntV data type

Type bits value	# of value bits	Value range	Total bits used
`0`	`0`	`0`	`1`
`10`	`3`	`0 to 0x07`	`5`
`110`	`6`	`0 to 0x3F`	`9`
`1110`	`16`	`0x to 0xFFFF`	`20`
`1111`	`32`	`0x to 0xFFFFFFFF`	`36`

Table 5.3 summarizes the value ranges for each variable length integer type in CDSN. The heading rows show the number of integer value bits for each value type bit combination, and the data cells show the range of integer values that can be stored for each data type.

Table 5.3 Value ranges for each variable length type

Variable Integer Type	Integer range for each value type bits value				
	Bits = 0 (0 value bits)	Bits = 10 (3 value bits)	Bits = 110 (6 value bits)	Bits = 1110 (16 value bits)	Bits = 1111 (32 value bits)
`UIntV`	0	0 to 7	0 to 63	0 to 65535	4,294,967,295
`IntV`	0	-4 to 3	-32 to 31	-32768 to 32767	-2,147,483,648 to 2,147,483,647
`UIntV16`	0	0 to 7	0 to 63	0 to 65535	N/A
`IntV16`	0	-4 to 3	-32 to 31	-32768 to 32767	N/A
`UIntV8`	0	0 to 7	0 to 63	N/A	N/A
`IntV8`	0	-4 to 3	-32 to 31	N/A	

Text Data Types

CDSN notation provides two data types: `Text` and `TextZ`.

The `Text` data type is used in CDSN notation to represent a string of characters, which can include a mix of 8-bit and 5-bit characters. This type conveniently masks the complexities of including escape and reset characters.

The combination of the `Tag` and `Text` types makes representing combinations of formatting and text sequences much easier. For example, the following text:

```
a cow
```

can be represented in CDSN Notation as:

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

The `TextZ` type is the `Text` type that always ends with a `cmlCharEnd` character. This type is most commonly used in tag parameter lists. For example, the format of the anchor tag is defined as:

```
Tag tag = cmlTagAnchor
TextZ name
```

In this specification, the `name` parameter is a string that holds the local anchor name. In the data stream, the string is followed by a `cmlCharEnd` character that delimits it from the following data.

NOTE: If a parameter is defined as type `TextZ`, the string value must end with the `cmlCharEnd` character.

PQA Tag Definitions

NOTE: Some tags have parameter values that must be included if another parameter has a certain value. Most of the time the second parameter is required if the value of the first parameter is `True`.

The parameter descriptions for these parameters state something like the following: "This parameter is required if the value of the `hasAlign` parameter is `True`." This means that the parameter is required if `hasAlign` is `True`, and is not expected in the input stream if `hasAlign` is `False`.

cmlTag8BitEncoding

Description Marks the beginning of 8-bit encoded text while in 5-bit encoding mode. This tag is only used within bit-packed data.

End Tag Delimited Yes

Parameters None

Example

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

cmlTagAddress

Description Delimits address data.

End Tag Delimited Yes

Parameters None

Example

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

cmlTagAnchor

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

Only use this tag to define local named anchors. Use the cmlTagHyperlink tag to define hyperlinks.

End Tag Delimited Yes

Parameters

`TextZ name`	String holding the local anchor name (not including the "#" character that precedes the anchor name in the HTML).

Example

```
Tag tag = cmlTagAnchor
TextZ name = "anchor"
```

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

Example

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

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
```

cmlTagCaption

Description Marks the caption to be placed above or below a table. It can appear anywhere in a table.

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
```

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 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 both margins are clear of images.

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

Example

```
Tag tag = cmlTagClear
Bit[2] clearAlign = cmlClearAll
```

cmlTagCMLEnd

Description Indicates the end of data for this resource.

End Tag Delimited No

Parameters None

Example

```
Tag tag = cmlTagCMLEnd
```

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.

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:

`cmlFlagFormIsLocalAction[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:).

`cmlFlagFormIsSecure[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 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.

`Bit post`

This parameter is required if the value of the `cmlFlagFormIsStandalone` parameter is `True`. If this value is 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`	This parameter is required if the value of the `cmlFlagFormIsStandalone` parameter is `True`. This is a 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`	This parameter is required if the value of the `cmlFlagFormIsStandalone` parameter is `True`. This is the 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
```

cmlTagH1

Description Marks a first level document heading.

End Tag Delimited Yes

Parameters

`Bit hasAlign`	A flag that is set if the `align` attribute is used.

`Bit[2] align`	This parameter is required if the value of the `hasAlign` parameter is `True`. This is an enumerated type that sets how the heading is aligned horizontally in the window. The value must be one of the following:
	`cmlAlignLeft`
	`cmlAlignCenter`
	`cmlAlignRight`

Example

```
Tag tag = cmlTagH1
Bit hasAlign = 1
Bit[2] align = alignCenter
Text "This is an H1 Heading"
Char cmlCharEnd // end heading tag
```

cmlTagH2

Description Marks a second level document heading.

End Tag Delimited Yes

Parameters

`Bit hasAlign`	A flag that is set if the `align` attribute is used.

`Bit[2] align`	This parameter is required if the value of the `hasAlign` parameter is `True`. This is an enumerated type that sets how the heading is aligned horizontally in the window. The value must be one of the following:
	`cmlAlignLeft`
	`cmlAlignCenter`
	`cmlAlignRight`

Example

```
Tag tag = cmlTagH2
```

```
Bit hasAlign = 1
Bit[2] align = alignCenter
Text "This is an H2 Heading"
Char cmlCharEnd // end heading tag
```

cmlTagH3

Description Marks a third level document heading.

End Tag Delimited Yes

Parameters

`Bit hasAlign`	A flag that is set if the `align` attribute is used.

`Bit[2] align`	This parameter is required if the value of the `hasAlign` parameter is `True`. This is an enumerated type that sets how the heading is aligned horizontally in the window. The value must be one of the following:
	`cmlAlignLeft`
	`cmlAlignCenter`
	`cmlAlignRight`

Example

```
Tag tag = cmlTagH3
Bit hasAlign = 1
Bit[2] align = alignCenter
Text "This is an H3 Heading"
Char cmlCharEnd // end heading tag
```

cmlTagH4

Description Marks a fourth level document heading.

End Tag Delimited Yes

Parameters

`Bit hasAlign`	A flag that is set if the `align` attribute is used.

`Bit[2] align`	This parameter is required if the value of the `hasAlign` parameter is `True`. This is an enumerated type that sets how the heading is aligned horizontally in the window. The value must be one of the following:
	`cmlAlignLeft`
	`cmlAlignCenter`
	`cmlAlignRight`

Example

```
Tag tag = cmlTagH4
Bit hasAlign = 1
Bit[2] align = alignCenter
Text "This is an H4 Heading"
Char cmlCharEnd // end heading tag
```

cmlTagH5

Description Marks a fifth level document heading.

End Tag Delimited Yes

Parameters

`Bit hasAlign`	A flag that is set if the `align` attribute is used.

`Bit[2] align`	This parameter is required if the value of the `hasAlign` parameter is `True`. This is an enumerated type that sets how the heading is aligned horizontally in the window. The value must be one of the following:
	`cmlAlignLeft`
	`cmlAlignCenter`
	`cmlAlignRight`

Example

```
Tag tag = cmlTagH5
Bit hasAlign = 1
Bit[2] align = alignCenter
Text "This is a Heading"
Char cmlCharEnd // end heading tag
```

cmlTagH6

Description Marks a sixth level document heading.

End Tag Delimited Yes

Parameters

`Bit hasAlign`	A flag that is set if the `align` attribute is used.

`Bit[2] align`	This parameter is required if the value of the `hasAlign` parameter is `True`. This is an enumerated type that sets how the heading is aligned horizontally in the window. The value must be one of the following:
	`cmlAlignLeft`
	`cmlAlignCenter`
	`cmlAlignRight`

Example

```
Tag tag = cmlTagH6
Bit hasAlign = 1
Bit[2] align = alignCenter
Text "This is an H6 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.

End Tag Delimited No

Parameters

`TextZ`	Null-terminated string value.

Example

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

cmlTagHorizontalRule

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

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.

`cmlFlagHRAlign[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 Viewer application.

`Uint16V size` This parameter is required if the value of the `cmlFlagHRCustom` parameter is `True`. This is the height (thickness) of the rule in pixels. The default is 1.

`Byte percent` This parameter is required if the value of the `cmlFlagHRIsPercent` parameter is `True`. This is the relative width of the rule in percentage of display width. The default is 100.

`Uint16V width` This parameter is required if the value of the `cmlFlagHRIsPercent` parameter is `False`. This is the 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
```

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. Use the `cmlTagAnchor` tag to define named anchors.

End Tag 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

1. By default, pages received from the Palm proxy server contain hash coded hyperlink indexes, instead of full URL specifications. In version 4.0 or later of the Palm OS, page designers can override this and send full URLs.

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.

`TextZ fragmentName`
This parameter is required if both the `cmlFlagLinkIsSameDoc` and `cmlFlagLinkIsFragment` parameter values are `True`. This is a string holding the fragment portion of the URL. For example, if the URL is `file:\foo.htm#section1`, then the fragment is `section1`.

`Uint16V PQFIndex`
This parameter is required if both the `cmlFlagLinkIsSameDoc` and `cmlFlagLinkInternal` parameter values are `True`. This is the index of the resource (referenced by the hyperlink) in the current PQA file. The first resource has an index of 1.

`TextZ href`
This parameter is required if the `cmlFlagInternal parameter` value is `False` and `cmlFlagHasHref` parameter values is `True`. This is a string holding the complete URL.

`Uint16V extLinkIndex`
This parameter is required if the `cmlFlagInternal` parameter value is `False` and `cmlFlagHasHref` parameter values is `False`. This is 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.

`TextZ title`
: This parameter is required if the value of the `cmlFlagLinkTitle` parameter is `True`. This is a string holding the title of the referenced page.

Example The following 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
```

The following 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, to conserve space, it does not include a URL or a title.

```
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
```

The following is an example of an internal link that is used to jump to another document within the same PQA: 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
```

cmlTagImage

Description Marks an image.

End Tag 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.

`Uint16V PQFLinkIndex`
: This parameter is required if the value of the `cmlFlagImageLocalPQA` parameter is `True`. This is the index of the image resource in the current PQA file. The first resource has an index of 1.

`TextZ alt`
: This parameter is required if the value of the `cmlFlagImageHasAlt` parameter is `True`. This is the alternate text string for the image.

`TextZ src`
: This parameter is required if the value of the `cmlFlagImageHasSrc` parameter is `True`. This is the source URL; only for references to resources in other (than the current) PQA files.

`Uint8V vSpace`
: This parameter is required if the value of the `cmlFlagImageHasVSpace` parameter is `True`. This is the vertical space between the image and the text above and below, in pixels, minus 1.

`Uint8V space`
: This parameter is required if the value of the `cmlFlagImageHasSpace` parameter is `True`. This is the horizontal space between the image and the text to the left and right, in pixels, minus 1.

`Uint8V border`
: This parameter is required if the value of the `cmlFlagImageHasBorder` parameter is `True`. This is the border width in pixels, minus 1.

`Bit[3] align`
: This parameter is required if the value of the `cmlFlagImageHasAlign` parameter is `True`. This is an enumerated type that sets how the image is aligned relative to the text. One of:

  `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.

`Image imageData`
: This parameter is required if the value of the `cmlFlagImageEmbedded` parameter is `True`. This is the 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
```

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

Viewer, 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.

`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.

`TextZ name` This parameter is required if the value of the `cmlFlagInputHasName` parameter is `True`. This is a string specifying the name of the checkbox.

`TextZ value` This parameter is required if the value of the `cmlFlagInputHasValue` parameter is `True`. This is a string holding the value for the checkbox. This value is sent to the server if the control is selected.

`TextZ Text` This parameter is required if the value of the `cmlFlagInputHasText` parameter is `True`. This is a 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
```

cmlTagInputDatePicker

Description Marks a date picker.

End Tag Delimited No

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.
`TextZ name`	This parameter is required if the value of the `hasName` parameter is `True`. This is a string holding the name of the date field.

Example

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

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 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.

`TextZ name` — This parameter is required if the value of the `cmlFlagInputHasName` parameter is `True`. This is a string holding the name of the input field.

`TextZ value` — This parameter is required if the value of the `cmlFlagInputHasValue` parameter is `True`. This is a string holding the initial value for the input field.

Example

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

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.

`TextZ name` — This parameter is required if the value of the `cmlFlagInputHasName` parameter is `True`. This is a string holding the name of the input field.

`TextZ value` — This parameter is required if the value of the `cmlFlagInputHasValue` parameter is `True`. This is a 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

`Uint16V 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 Viewer, 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.

`TextZ name` — This parameter is required if the value of the `cmlFlagInputHasName` parameter is `True`. This is a string holding the name of the radio button control.

`TextZ value` — This parameter is required if the value of the `cmlFlagInputHasValue` parameter is `True`. This is a string holding the value for the radio button. This value is sent to the server if the control is selected.

`TextZ Text` — This parameter is required if the value of the `cmlFlagInputHasText` parameter is `True`. This is a 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:"
```

cmlTagInputReset

Description Marks a reset button in a form.

End Tag Delimited No

Parameters

`Bit hasValue`	Set if the `hasValue` attribute is used to set a custom button label.
`TextZ value`	This parameter is required if the value of the `hasName` parameter is `True`. This is a 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"
```

cmlTagInputSubmit

Description Marks a submit button in a form.

End Tag 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.

`TextZ name`	This parameter is required if the value of the `cmlFlagInputHasName` parameter is `True`. This is a string holding the name of the button.
`TextZ value`	This parameter is required if the value of the `cmlFlagInputHasValue` parameter is `True`. This is a 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"
```

cmlTagInputTextArea

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

End Tag 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.
`TextZ name`	This parameter is required if the value of the `cmlFlagInputHasName` parameter is `True`. This is a 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 \nhere: "
Char cmlCharEnd
```

cmlTagInputTextLine

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

End Tag 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.

`TextZ name`
: This parameter is required if the value of the `cmlFlagInputHasName` parameter is `True`. This is a string holding the name of the input field.

`TextZ value`
: This parameter is required if the value of the `cmlFlagInputHasValue` parameter is `True`. This is a string holding the initial value for the input field.

Example

```
Tag tag = cmlTagForm
Text "Enter Name:"

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

cmlTagInputTimePicker

Description Marks a time picker.

End Tag 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.
`TextZ name`	This parameter is required if the value of the `hasName` parameter is `True`. This is a string holding the name of the time field.

Example

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

cmlTagLinkColor

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

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
```

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

```
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
```

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 No

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.

`Uint16V value` This parameter is required if the value of the `cmlFlagListModValue` parameter is `True`. This is ignored for unordered lists. In ordered lists, `value` is the numeric value for this element, minus 1.

`Bit[3] type` This parameter is required if the value of the `cmlFlagListModType` parameter is `True`. This is 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
Uint16V value = 0
Text "Third item"
```

cmlTagListItemDefinition

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

End Tag Delimited No

Parameters None

Example

```
Tag tag = cmlTagListItemDefinition
Text "Definition of term."
```

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 Delimited No

Parameters None

Example

```
Tag tag = cmlTagListItemNormal
Text "Third item"
```

cmlTagListItemTerm

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

End Tag Delimited No

Parameters None

Example

```
Tag tag = cmlTagListItemTerm
Text "Term for definition"
```

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 Delimited 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 Delimited Yes

Parameters

`Bit[3] type` — An enumerated type that specifies the bullet type. One of:

`cmlListTDisc`
: Filled circle bullet

`cmlListTSquare`
: Filled square bullet

`cmlListTCircle`
: Open circle bullet

Example

```
// 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
```

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."
```

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.

`TextZ name`
: This parameter is required if the value of the `cmlFlagInputHasName` parameter is `True`. This is a 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
```

cmlTagSelectItemCustom

Description Precedes a custom item in a selection menu.

End Tag Delimited No

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.

`TextZ value`	This parameter is required if the value of the `cmlFlagInputHasName` parameter is `True`. This is 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

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

cmlTagSelectItemNormal

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.

End Tag Delimited No

Parameters None

Example

```
Tag tag = cmlTagSelectItemNormal
TextZ "Third item"
```

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 `cmlTagTableData` 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 Yes

Parameters

`Bit[7] flags` 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.

`Bit[2] hAlign` This parameter is required if the value of the `cmlFlagTableHasAlign` parameter is `True`. This is an enumerated type setting how the table is aligned on the page. This must be one of the following values:

`cmlAlignLeft`

`cmlAlignCenter`

`cmlAlignRight`

`Uint16V width` This parameter is required if the value of the `cmlFlagTableHasWidth` parameter is True. This is the table width in pixels. 0 indicates to calculate the width of the table is from the contents.

`Uint8V border` This parameter is required if the value of the `cmlFlagTableHasBorder` parameter is `True`. This is the border width in pixels. 0 indicates to suppress the border.

`Uint8V cellSpacing`

This parameter is required if the value of the `cmlFlagTableHasCellSpacing` parameter is `True`. This is the 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.

`Uint8V cellPadding`

This parameter is required if the value of the `cmlFlagTableHasCellPadding` parameter is `True`. This is the 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
```

cmlTagTableData

Description Marks a data cell in a table. 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 `Bit[2] hAlign` This parameter is required if the value of the `cmlFlagCellHasAlign` parameter is `True`. This is an enumerated type that sets horizontal cell alignment. The must be one of the following values:

`cmlAlignLeft`

`cmlAlignCenter`

`cmlAlignRight}`.

`Bit[2] vAlign` This parameter is required if the value of the `cmlFlagCellHasVAlign` parameter is `True`. This is an enumerated type that sets vertical cell alignment. The must be one of the following values:

`cmlVAlignTop`

`cmlVAlignCenter`

`cmlVAlignBottom`

`Uint8V colSpan`

This parameter is required if the value of the `cmlFlagCellHasColSpan` parameter is `True`. This is the number of columns spanned by the cell, minus 1. For example, if the cell spans one column, this is set to 0.

`Uint8V rowSpan`

This parameter is required if the value of the `cmlFlagCellHasRowSpan` parameter is `True`. This is the number of rows spanned by the cell, minus 1.

`Uint16V height`

This parameter is required if the value of the `cmlFlagCellHasHeight` parameter is `True`. This is the height of the cell in pixels.

`Uint16V width` This parameter is required if the value of the `cmlFlagCellHasWidth` parameter is `True`. This is the 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 `Bit[2] hAlign` This parameter is required if the value of the `cmlFlagCellHasAlign` parameter is `True`. This is an enumerated type that sets horizontal cell alignment. The must be one of the following values:

`cmlAlignLeft`

`cmlAlignCenter`

`cmlAlignRight`

`Bit[2] vAlign` This parameter is required if the value of the `cmlFlagCellHasVAlign` parameter is `True`. This is an enumerated type that sets vertical cell alignment. The must be one of the following values:

`cmlVAlignTop`

`cmlVAlignCenter`

`cmlVAlignBottom`

`Uint8V colSpan`

This parameter is required if the value of the `cmlFlagCellHasColSpan` parameter is `True`. This is the number of columns spanned by the cell, minus 1. For example, if the cell spans one column, this is set to 0.

`Uint8V rowSpan`

This parameter is required if the value of the `cmlFlagCellHasRowSpan` parameter is `True`. This is the number of rows spanned by the cell, minus 1.

`Uint16V height`

This parameter is required if the value of the `cmlFlagCellHasHeight` parameter is `True`. This is the height of the cell in pixels.

`Uint16V width` This parameter is required if the value of the `cmlFlagCellHasWidth` parameter is `True`. This is the 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
```

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.

`Bit[2] hAlign`	This parameter is required if the value of the `hasAlign` parameter is `True`. This is an enumerated type that sets how text is aligned horizontally within the cells in the row. This value must be one of the following:
	`cmlAlignLeft`
	`cmlAlignCenter`
	`cmlAlignRight`

`Bit[2] vAlign`	This parameter is required if the value of the `hasAlign` parameter is `True`. This is an enumerated type that sets how text is aligned vertically within the cells in the row. This value must be one of the following:
	`cmlVAlignTop`
	`cmlVAlignCenter`
	`cmlVAlignBottom`

Example

```
Tag tag = cmlTagTableRow
Bit hasAlign = 0
Tag tag = cmlTagTableHeader
Bit[7] flags = 0
Text "row1, col1"
Char cmlCharEnd

Tag tag = cmlTagTableRow
Bit hasAlign = 1
Bit[2] hAlign = cmlAlignRight
Bit[2] vAlign = cmlVAlignTop
Tag tag = cmlTagTableData
Bit[7] flags = 0
Text "row2, col1"
Char cmlCharEnd
```

cmlTagTextBold

Description Marks bold text style.

End Tag Delimited Yes

Parameters None

Example

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

cmlTagTextColor

Description Sets the text color.

End Tag Delimited No

Parameters

`Byte red`	A value from 0 to 255 that indicates the amount of red in the RGB color specification
`Byte green`	A value from 0 to 255 that indicates the amount of green in the RGB color specification.
`Byte blue`	A value from 0 to 255 that indicates the amount of blue in the RGB color specification.

Example

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

Text "This text is reddish"
```

cmlTagTextItalic

Description Marks italic text style.

End Tag Delimited Yes

Parameters None

Example

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

cmlTagTextMono

Description Marks monospace text style.

End Tag Delimited Yes

Parameters None

Example

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

cmlTagTextSize

Description Sets the current text size.

End Tag Delimited No

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

Example

```
Tag tag = cmlTagTextSize
Bit[3] size = 3
```

cmlTagTextStrike

Description Marks strike-through text style.

End Tag Delimited Yes

Parameters None

Example

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

cmlTagTextSub

Description Marks subscript text style.

End Tag Delimited Yes

Parameters None

Example

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

cmlTagTextSup

Description Marks superscript text style.

End Tag Delimited Yes

Parameters None

Example

```
// Start superscript text
```

```
Tag tag = cmlTagTextSup
Text "This is superscript text"
// End superscript text
Char  end = cmlCharEnd
```

cmlTagTextUnderline

Description Marks underlined text style.

End Tag Delimited Yes

Parameters None

Example

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

Summary of CML Tags

The following table categorizes the CML tags.

CML Tags	
Background Attributes	
cmlTagBGColor	
Forms	
cmlTagForm	cmlTagInputSubmit
cmlTagInputCheckBox	cmlTagInputTextArea
cmlTagInputDatePicker	cmlTagInputTextLine
cmlTagInputHidden	cmlTagInputTimePicker
cmlTagInputPassword	cmlTagSelect
cmlTagInputRadio	cmlTagSelectItemCustom
cmlTagInputReset	cmlTagSelectItemNormal
Graphical Elements	
cmlTagHorizontalRule	cmlTagImage

CML Tags *(continued)*

Hyperlinks

cmlTagAnchor
cmlTagHyperlink

Lists

cmlTagListDefinition
cmlTagListItemCustom
cmlTagListItemDefinition
cmlTagListItemNormal
cmlTagListItemTerm
cmlTagListOrdered
cmlTagListUnordered

Other Elements

cmlTagClear
cmlTagCMLEnd

Paragraph Attributes

cmlTagAddress
cmlTagBlockQuote
cmlTagParagraphAlign

Tables

cmlTagCaption
cmlTagTable
cmlTagTableData
cmlTagTableHeader
cmlTagTableRow

Text Attributes

cmlTag8BitEncoding
cmlTagH1
cmlTagH2
cmlTagH3
cmlTagH4
cmlTagH5
cmlTagH6
cmlTagHistoryListText
cmlTagLinkColor
cmlTagTextBold
cmlTagTextColor
cmlTagTextItalic
cmlTagTextMono
cmlTagTextSize
cmlTagTextStrike
cmlTagTextSub
cmlTagTextSup
cmlTagTextUnderline

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W