



Adobe CMap and CIDFont Files Specification

Adobe Developer Support

Technical Specification #5014

Version 1.0

16 October 1995

Adobe Systems Incorporated

Corporate Headquarters
1585 Charleston Road PO Box 7900
Mountain View, CA 94039-7900
(415) 961-4400 Main Number
(415) 961-4111 Developer Support
Fax: (415) 969-4138

Adobe Systems Benelux B.V.
Europlaza
Hoogoorddreef 54a
1101 BE Amsterdam Z.O.
The Netherlands
+31-20-6511 355
Fax: +31-20-6511 313

Adobe Systems Eastern Region
24 New England
Executive Park
Burlington, MA 01803
(617) 273-2120
Fax: (617) 273-2336

Adobe Systems Co., Ltd.
Yebisu Garden Place Tower
4-20-3 Ebisu, Shibuya-ku
Tokyo 150
Japan
+81-3-5423-8169
Fax: +81-3-5423-8204

© 1993, 1994 Adobe Systems, Incorporated. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of the publisher. Any software referred to herein is furnished under license and may only be used or copied in accordance with the terms of such license.

PostScript is a trademark of Adobe Systems Incorporated. All instances of the name PostScript in the text are references to the PostScript language as defined by Adobe Systems Incorporated unless otherwise stated. The name PostScript also is used as a product trademark for Adobe Systems' implementation of the PostScript language interpreter.

Any references to a "PostScript printer," a "PostScript file," or a "PostScript driver" refer to printers, files, and driver programs (respectively) which are written in or support the PostScript language. The sentences in this book that use "PostScript language" as an adjective phrase are so constructed to reinforce that the name refers to the standard language definition as set forth by Adobe Systems Incorporated.

PostScript, the PostScript logo, Display PostScript, Adobe, the Adobe logo, Adobe Type Manager, Adobe Type Manager-Japanese Edition, ATM, Display PostScript, and Poetica are trademarks of Adobe Systems Incorporated registered in the U.S.A. and in other jurisdictions. FutoGoB101, FutoM-inA101, Jun101, Ryumin Light KL, Gothic BBB Medium, and Skiksei Kaisho CBSK1 are trademarks of Morisawa and Company, Ltd. Apple and Macintosh are registered trademarks of Apple Computer, Inc. Microsoft and Windows are registered trademarks of Microsoft, Inc. Fujitsu is a registered trademark of Fujitsu Limited. NEC is a registered trademark of NEC Information Systems, Inc. Other brand or product names are the trademarks or registered trademarks of their respective holders.

This publication and the information herein is furnished AS IS, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies, makes no warranty of any kind (express, implied or statutory) with respect to this publication, and expressly disclaims any and all warranties of merchantability, fitness for particular purposes and noninfringement of third party rights.



Contents

List of Figures v

List of tables vii

Adobe CMap and CIDFont Files Specification 1

- 1 Introduction 1
 - Compatibility 1
 - Copyrights for CID-Keyed Font Programs 2
 - Overview 2
- 2 CMap and CIDFont Resource Architecture 3
 - Terminology 3
 - Native Support Versus Compatibility Mode 4
 - The Character Collection 5
 - Version Control 5
 - The CIDFont File 6
 - The CMap File 6
- 3 CIDFont Tutorial 8
 - CIDFont File Components 8
 - CIDFont Example 10
- 4 CIDFont Reference 26
 - CIDFont Organization 26
 - CIDFont Resource Keys 27
 - Defining the CIDFont Resource 31
- 5 CMap Tutorial 32
 - CMap File Components 33
 - First Example: Stand-Alone CMap File 33
 - Closing the CMap File and Creating the Resource Instance 45
 - Second Example: A CMap File That Uses Another 46
 - CMap File Naming Convention 48
- 6 Rearranged Font Tutorial 48
 - Rearranged Font Components 49
 - Rearranged Font Example 50
- 7 CMap Reference 59
 - CMap File Nomenclature and Lexical Elements 60
 - Operator Summary 62

CMap File Overview 63
Operator Details 63

**Appendix A:
Installing CID-Keyed Fonts
on PostScript Interpreters 73**

- A.1 Introduction 73
- A.2 PostScript Interpreter Requirements 73
- A.3 Categories of Installation Files 74
- A.4 Installation Environment 74
- A.5 Prior to Installation 77
- A.6 Installation of Category A Files 78
- A.7 Installation of Category B Files 80

**Appendix B:
ATM-J Compatibility
for CID-Keyed Fonts 81**

- B.1 Installing CID-Keyed Fonts on the Macintosh 81
 - CIDFont Files 81
 - CMap Files 82
- B.2 Naming Conventions 82
- B.3 Parsing Considerations 82
- B.4 Miscellaneous Notes for Macintosh ATM before version 3.5 84
- B.5 Miscellaneous Notes for Macintosh ATM version 3.5 84

**Appendix C:
Obtaining
CID Information 85**

- C.1 Support for CID-keyed Font Development 85

**Appendix D:
Font Naming and Unique ID Numbers 87**

- D.1 CID Font Naming 87
- D.2 Calculating Unique IDs 88
 - Assigning the ID Count 88
- D.3 Miscellaneous Notes 90

**Appendix E:
Changes since Earlier Versions 91**

- E.1 Changes in the 16 October 1995 version: 91

Index 93



List of Figures

- Figure 1 Character Codes to CIDs and Glyphs 6
- Figure 2 CIDMap, FDArray, and charstring data 10
- Figure 3 Internal organization of the CIDMapOffset string 20
- Figure 4 Empty intervals 22
- Figure 5 Relationship of SubrMap to subroutine data length 25
- Figure 6 Codespace ranges for the 83pv-RKSJ-H charset encoding 41



List of tables

Table 1	Relationship of input code to selector	61
Table 2	PostScript language lexical elements	61
Table C.1	Whom to contact at Adobe Systems	85
Table D.1	UIDOffset values	89

Adobe CMap and CIDFont Files Specification

1 Introduction

Character codes and character names are both widely used in PostScript™ language programs to access font glyphs. This document introduces another character-access type, the *character identifier*, abbreviated as *CID*. This document explains what a CID is, and describes the files that use CIDs. These files are used together to produce a font called a *CID-keyed font*, so named because the glyphs are accessed by CID.

This section describes the compatibility issues for CID-keyed fonts, explains that CID-keyed fonts are copyrightable, and provides an overview for the rest of the document. After reading this section, you should be ready to start learning about CID-keyed font files and how they are used.

Note This version of this document, dated 16 October 1995, provides a completely rewritten version of Appendix A — Installing CID-Keyed Fonts on a PostScript Interpreter. Other minor changes are noted in Appendix E.

1.1 Compatibility

The PostScript interpreter has undergone continual enhancement since its debut in late 1984. During this time, Adobe Systems has changed both the PostScript interpreter implementation and the features of font formats. These changes are generally compatible with all versions of the PostScript interpreter. Features introduced by this specification are likewise compatible.

There are several parts of this document dealing with compatibility concerns. In particular, Appendix A, “Installing CID-Keyed fonts on PostScript Interpreters,” describes how CID-keyed font files are installed for use with both embedded interpreters such as those found in printers and imagesetters, as well as with host-based interpreters such as DPS (Display PostScript) and CPSI (Configurable PostScript Interpreter). Appendix B, “ATM™-J Compatibility with CID-Keyed Fonts,” describes how CID-keyed font files are installed for use with the Adobe Type Manager™ product, Japanese edition.

Any future extensions to Adobe™ CID-keyed font files will be designed so that those extensions can be ignored by the current generation of interpreters. New extensions will often take the form of new dictionary entries; other extensions may define additional procedures. As long as interpreters for CID-keyed font software are written to ignore such possible future extensions, correct font interpretation will result. Future extensions will be thoroughly described in revisions of this document.

Some CID-keyed font rendering software (such as ATM-J) takes advantage of a particular stylized use of the PostScript language. As a result, CID-keyed font files must also adhere to these PostScript language usage conventions. The syntax resulting from these conventions is considerably more restricted than that of the PostScript language; CID-keyed fonts can be read and executed by PostScript interpreters, but not all PostScript language usage is acceptable in CID-keyed fonts. These restrictions will be noted wherever necessary in this document, particularly in Appendices A and B.

1.2 Copyrights for CID-Keyed Font Programs

Because CID-keyed fonts are computer programs, they are copyrightable to the same extent as other computer software. The ideas expressed by copyrighted works are not protected; however, the particular expression is. In the case of CID-keyed font programs, this means that while the typeface shapes are not protected, the program text is.

Unauthorized duplication of a CID-keyed font program is a violation of copyright law. Such unauthorized activities include verbatim copying and distribution, as well as less obvious activities such as modification and translation of the program from one form or format into another.

Adobe Systems' CID-keyed font programs are licensed for use on one or more devices (depending on the terms of the particular license). These licenses generally permit the use of a licensed program in a system that translates a CID-keyed font program into some other format in the process of rendering, as long as a copy of the program (even in translated form) is not produced.

The personal computer industry and its customers have benefitted greatly from copyright protection. Copyright protection gives the developer of a CID-keyed font program the incentive to create excellent typeface programs. In turn, the user of CID-keyed font programs can expect to have available the finest typeface software to choose from.

1.3 Overview

The remaining chapters of this document summarize the various components of a CID-keyed font and how they work together.

- Section 2 provides an overview of the CID-keyed font architecture.
- Section 3 explains how the component CIDFont is put together.
- Section 4 is a reference section of CIDFont operators and syntax.
- Section 5 discusses how the component CMap is built.
- Section 6 discusses producing rearranged fonts.
- Section 7 is a reference section of CMap operators and syntax.
- Appendix A provides details for installing CID-keyed fonts on PostScript interpreters such as printers and DPS.
- Appendix B provides details for installing CID-keyed fonts on a host for use with ATM-J.
- Appendix C provides information on getting a registry and vendor registration, unique IDs, and other useful technical notes from Adobe Developer Relations.
- Appendix D provides information on naming conventions and how to utilize UniqueID numbers for CID-keyed fonts.
- Appendix E lists changes since earlier versions of this document.

2 CMap and CIDFont Resource Architecture

This section provides a conceptual overview of CMaps and CIDFonts. After reading this section, you should understand the terms to be used in this document and know what CMaps and CIDFonts are and how they interact.

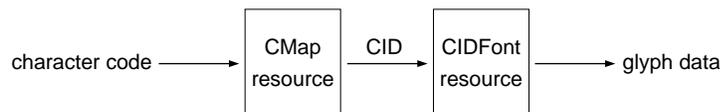
2.1 Terminology

A *character* is an abstract notion denoting a class of shapes declared to have the same meaning or form. A *glyph* is a specific instance of a character. For example, consider the class of shapes named “ampersand” and “fi ligature” along with a few instances of each class:

<i>Character</i>	<i>Glyphs</i>
Class of Shape	Sample Instances of the Character
ampersand	& & & & & & & ...
fi ligature	fi fi fi fi fi fi ...

A *character collection*, another abstract notion, is a collection or group of distinct characters. A *character identifier*, or *CID*, is a concrete notion in which an integer is associated with a character from a character collection. When the characters in a character collection are distinctly numbered with CIDs from 0 to $n - 1$ for a character collection of n characters, the character collection is called an *ordered character collection*.

A *character code* is that portion of a string used by **show** (or other similar operator) that corresponds to a character. A *CID-keyed font* is a font program that maps character codes to CIDs, and uses CIDs to access glyph data. There are two parts to a CID-keyed font: a CMap resource and a CIDFont resource. The CMap, or character code map, maps character codes to glyph selectors. For CIDFonts, this selector is a CID. The CIDFont uses CIDs to access glyph data. These components are used to access glyph data as the following diagram depicts:



The CMap can also map character codes to two other glyph selector types. The first is a character code and can occur when the font resource is other than CIDFont. The second is a *character name*, a PostScript language name object that uniquely identifies a character, and can also occur when the font resource is other than CIDFont.

Note that a CMap specifies a subset of a character collection to be used, called a *character set*, or *charset*. In addition, the CMap imposes an encoding on that subset. A font resource can be referenced by different CMaps, each of which defines a different charset and encoding. Likewise, many font resources can be referenced by a single CMap, accessing different glyphs for the same character instantiated in each font resource.

2.2 Native Support Versus Compatibility Mode

This document introduces a new set of PostScript language commands (procedures or operators) that are defined in a procset resource. PostScript interpreters that have built-in support for these commands are considered to provide *native-support* for font programs that use them. Other PostScript interpreters can be provided with PostScript language procedures that emulate the same outward behavior of these commands. These interpreters are said to be operating in *compatibility mode*. At the time of this writing, compatibility mode supports only the CID glyph selector for CID-keyed font programs.

A *file* is an external representation of a resource, such as a CMap program or a CIDFont program, and is distinct from the internal virtual memory (VM) representation that results when such a file is parsed by a font interpreter. While both native-support interpreters and those operating in compatibility mode use the same CMap and CIDFont files, the structures created in VM may be very different. Native-support interpretation of CMap and CIDFont resource files materialize more-or-less directly as dictionary objects in VM, which the PostScript interpreter uses directly. In compatibility mode, execution of the CIDMap and CIDFont files results in the construction of a composite font hierarchy, which bears little resemblance to the structure of the CMap and CIDFont files and whose structure is undocumented. For more information, read Appendix A, *Installing CID-Keyed Fonts on PostScript Interpreters*.

2.3 The Character Collection

The first step in building CID-keyed fonts is to decide on the members of a character collection, and impose an order on them. The CIDs that identify the members of a character collection are used to order the collection. Hereafter, assume *character collection* means *ordered character collection*.

Note A CID-keyed font must be based on one and only one character collection. All CID-keyed fonts based on a particular character collection use identical CID index values to access corresponding glyph data.

The CID index value of 0 is always used to refer to the character meaning “the undefined or ‘notdef’ character.” This CID is used when the CMap file does not explicitly indicate a mapping for a character code.

2.4 Version Control

Both the CIDFont and the CMap must use CIDs from compatible character collections. The identification of the character collection is accomplished by placing version control information into each CIDFont and CMap file. To identify a character collection uniquely, three components are needed:

- a *registry* name is used to identify an issuer of orderings;
- an *ordering* name is used to identify an ordered character collection; and,
- a *supplement* number is used to indicate that the ordered character collection for a registry, ordering, and *previous* supplement has been changed to add *new* characters assigned CIDs beginning with the next available CID.

These three pieces of information taken together uniquely identify a character collection. In a CIDFont, this information declares what the character collection is. In a CMap, this information specifies which character collection is

required for compatibility. A CMap is compatible with a CIDFont if the registry and ordering are the same. If the supplement numbers are different, some codes may map to the CID index of 0. Details about how this version information is specified and its impact on CIDFont and CMap files are found in the sections that follow.

2.5 The CIDFont File

The CIDFont file contains glyph data that are indexed by CID. If the CIDFont file is missing glyph data for a particular CID, the CID with an index value of 0 (which must have glyph data) is used.

The CIDFont file contains character instances, or glyphs. In the example below, note that CID 7 refers to different shapes in the CIDFonts, but always means “ampersand.” Likewise, CID 112 refers to another class of shapes, but always means “fi ligature.”

Character	CID	CIDFont 1	CIDFont 2	CIDFont 3
ampersand	7	&	&	<i>ě</i>
fi ligature	112	fi	fi	<i>fi</i>

2.6 The CMap File

The CMap file is used to determine which CID is referenced by a particular character code. Many CMap files can be used with a CIDFont file. Each CMap file specifies a particular subset of the character collection, the *charset*, that it will use. Various subsets of a character collection may be wanted for several reasons, for example:

- Different platform vendors have defined their own system-specific character sets. By producing a character collection of the union of all character sets, CID-keyed fonts are portable across different platforms.
- Variations of a font are needed. For example, in Japanese or Chinese text, writing may be horizontal or vertical.

The following figure shows some sample character codes, the corresponding CIDs that result when the character codes are translated by two CMaps, and the glyphs associated with the CIDs for two CIDFonts.

Figure 1 *Character Codes to CIDs and Glyphs*

Code	CMap 1 83pv-RKSJ-H			CMap 2 Ext-RKSJ-V		
	CID	CIDFont 1 and 2		CID	CIDFont 1 and 2	
<82A8>	851	お	お	851	お	お
<57>	56	W	W	286	W	W
<8179>	690	【	【	7915	ㄣ	ㄣ
<817A>	691	】	】	7916	ㄣ	ㄣ
<8D7B>	2030	礪	礪	5853	礪	礪
<E1E6>	5853	礪	礪	2030	礪	礪
<92CD>	3051	搦	搦	7747	搦	搦
<81F6>	777	𠄎	𠄎	0		

The row with character code <82A8> represents the most typical situation in which two CMap files refer to the same data. Most CMap files for a character collection differ in relatively few mappings of character codes to CIDs.

The row with character code <57> illustrates a difference between two CMap files based on the platform. The CMap 1, 83pv-RKSJ-H, intended for use on Macintosh platforms, uses proportionally spaced Roman characters, while the CMap 2, Ext-RKSJ-V, intended for use on PC platforms, uses half-width Roman characters.

The rows with character codes <8179> and <817A> illustrate where variations of a font are required. CMap 1 is used to access the horizontally written characters from a font, while CMap 2 is used to access those that are written vertically.

The rows with character codes <8D7B> and <E1E6> demonstrate how characters are swapped depending on the platform. This typically occurs when old-style characters are to be superseded, but the old-style characters are yet to be maintained in the charset, though not in the primary character code position. The row with character code <92CD> shows how characters can be replaced.

The row with character code <81F6> demonstrates that CMap files can map character codes to a notdef character. While in CMap 1 the character code <81F6> maps to the “double dagger” character (CID 777 in the example), the same character code maps to the default notdef character (CID 0) in CMap 2. In both CIDFont examples shown, the default notdef character is the same as the “full-width space” character, with glyphs consisting of horizontal dimension only. These CIDFonts could just as well have used any other glyph as instances of the default notdef character.

Note For information on the the CMap files for the Japanese language group to which specific characters map, obtain the document CID-Keyed Japanese Font Glyph Complement, Adobe Technical Note #5078, as is listed in Appendix C.

3 CIDFont Tutorial

This section describes CIDFont files from the perspective of the font developer who wishes to build a character collection in the form of a CIDFont file. While several files comprise a complete CID-keyed font, font vendors primarily interested in supporting the standard character sets and encodings of the Japanese language group need only develop the CIDFont file.

After reading this section, you should be able to understand the example and use it, along with other sections from this document, as a starting point to construct different CIDFont files.

3.1 CIDFont File Components

As explained in section 2, a CIDFont file is a PostScript language font resource specifically designed to accommodate a large collection of characters, and may have imposed on it diverse encoding requirements representing one or more character sets within the collection.

CIDFont files *are like* other PostScript font resources in the following ways:

- CIDFont files are PostScript language programs that adopt a restrictive syntax—as is the case with Type 1 font programs.
- CIDFont files contain collections of traditional Type 1 or Type 3 character descriptions and the hinting information needed to rasterize them.
- The CIDFont files have a font type. The fonts described in this document are of CIDFontType 0. Other CIDFontType designations are reserved.
- CIDFont files can be used from disk or ROM, or loaded into VM.

CIDFont resources *differ* from other types of PostScript font resources in the following ways:

- Glyph data (also called *character descriptions* or *charstring data*) in CIDFonts are always referenced using character IDs.
- Encoding information is described in the CMap file—not in the character collection.

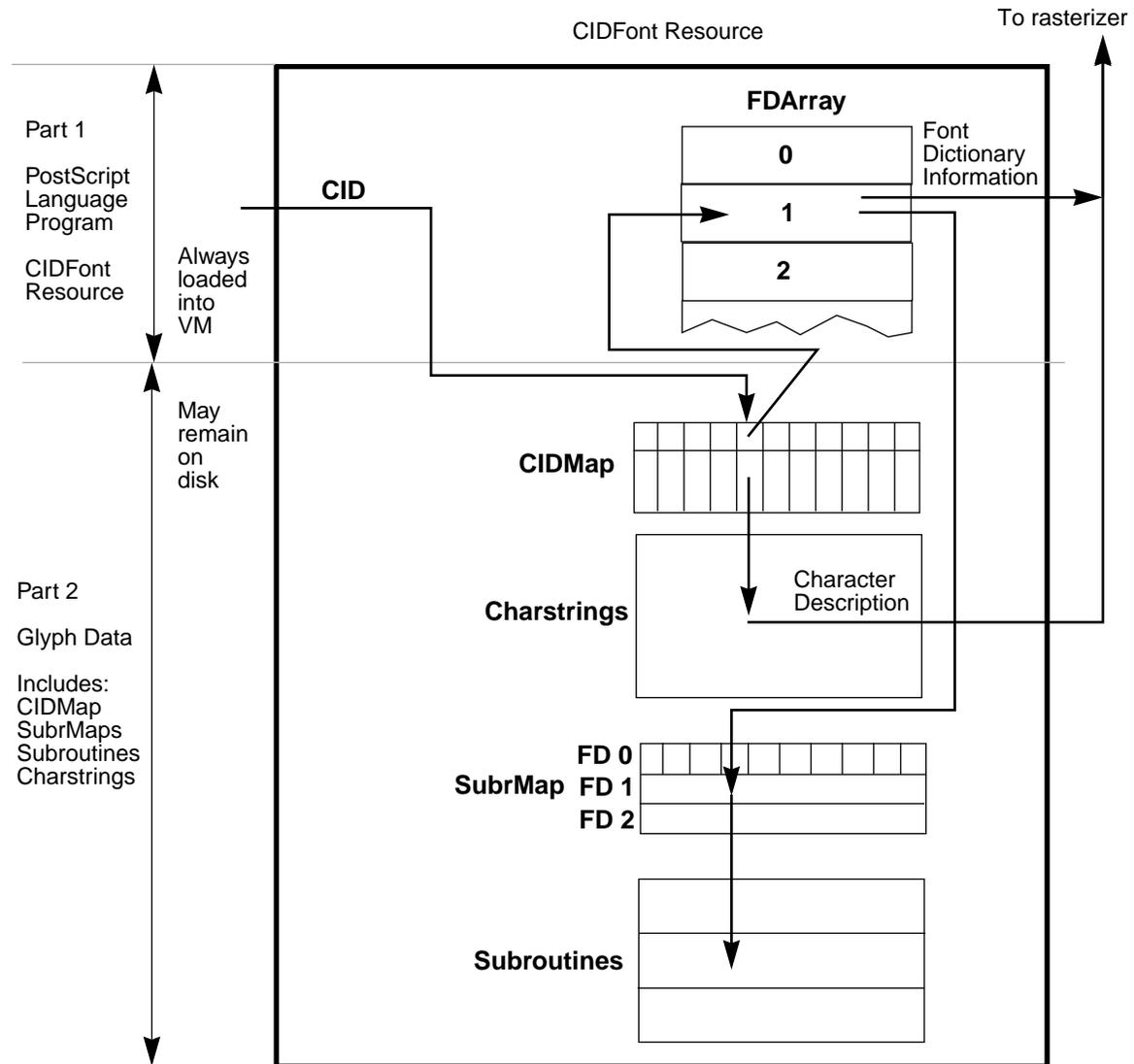
Because the exact VM representation of CIDFonts and the mechanism by which they are created and used may change over time, the CIDFont and CMap file strategy outlined here intentionally separates font development from the creation of composite font structures in VM. Composite font structures become a function of special operators and procsets supplied by Adobe; the developer is freed to enhance the fonts themselves.

A CIDFont file consists of two parts. Part one is a PostScript language program that defines a *CIDFont resource instance*. Part two is a collection of glyph data along with some additional data. The underlying type of the resource instance is a dictionary object.

Part two, the glyph data, either resides in a file system or in VM. The file system can be disk-based, ROM-based, or cartridge-based; such forms vary only in regard to performance issues. This document generally assumes a file system format that is disk-based for glyph data, but other formats are possible and even likely. Where important, differences from other formats are noted.

Figure 2 is a data flow diagram of the internal organization of a CIDFont. In VM, the CMap resource produces a character ID for use by the CIDFont resource. The character ID acts as an index into the CIDMap, which is in turn used to locate other pieces of information. Each interval of the CIDMap also has two parts. The first part is an index into the FDArray, which is an array of font dictionaries. The second part is an offset into the charstring data. Charstring data, subroutine information (if any), and data from the appropriate member of the FDArray of font dictionaries, are required to rasterize a glyph.

Figure 2 CIDMap, FDArray, and charstring data



3.2 CIDFont Example

This section presents a CIDFont example, including a font dictionary in the FDArray. The example is first given in full, and then is analyzed in detail in the sections that follow. Where statements or data have been omitted, they are replaced with explanatory text within brackets like this:

<< text here omitted >>

A CIDFont file is a program written in the PostScript language. Section 4 explains the syntax, and tells which entries are required and which are optional. The ordering of the key-value pairs in the dictionary portion of the

file (the part loaded into VM) is unimportant; in the portion of the file that usually remains on disk (charstrings, subroutines, and their offset maps), offset information is very important. Because the data section is offset-based, do not alter this section of a CIDFont resource casually—you may risk making hundreds of offsets incorrect.

Example 1: *Example CIDFont file, including font dictionary*

```

%!PS-Adobe-3.0 Resource-CIDFont
%%DocumentNeededResources: procset CIDInit
%%IncludeResource: procset CIDInit
%%BeginResource: CIDFont Ryumin-Light
%%Title: (Ryumin-Light Adobe Japan1 0)
%%Version: 1

/CIDInit /ProcSet findresource begin

20 dict begin

/CIDFontName /Ryumin-Light def
/CIDFontVersion 1 def
/CIDFontType 0 def

/CIDSystemInfo 3 dict dup begin
  /Registry (Adobe) def
  /Ordering (Japan1) def
  /Supplement 0 def
end def

/FontBBox [-180 -293 1090 1010] def

/UIDBase 27611 def
/XUID [1 11 27611] def

/FontInfo 2 dict dup begin
  /Notice ((c) Copyright 1993 Adobe Systems Incorporated. All
  Rights Reserved.) def
  /FullName (Ryumin-Light) def
end def

/CIDMapOffset 0 def
/FDBytes 1 def
/GDBytes 3 def
/CIDCount 8284 def

/FDArray 3 array

dup 0
  %ADOBeginFontDict
  14 dict begin

  /FontName /Ryumin-Light-Proportional def
  /FontType 1 def
  /FontMatrix [ 0.001 0 0 0.001 0 0 ] def
  /PaintType 0 def

  %ADOBeginPrivateDict

```

```

/Private 25 dict dup begin
  /MinFeature {16 16} def
  /lenIV 1 def
  /LanguageGroup 1 def
  /BlueValues [-14 0 662 682 448 458] def
  /BlueScale 0.0396271 def
  /BlueFuzz 1 def
  /BlueShift 7 def
  /StdHW [85] def
  /StdVW [85] def
  /StemSnapH [85] def
  /StemSnapV [85] def

  /OtherSubrs
  [ {} {} {}
    { systemdict /internaldict known not
      { pop 3 }
      { 1183615869 systemdict /internaldict get exec dup
        /startlock known
        { /startlock get exec }
        { dup /strlck known
          { /strlck get exec }
          { pop 3 }
          ifelse
        }
        ifelse
      }
      ifelse
    } bind
    {} {} {} {} {} {} {} {}
    { 2 { cvi { { pop 0 lt { exit } if } loop } repeat }
      repeat } bind
  ] def

  /password 5839 def

  /SubrMapOffset 33140 def
  /SDBytes 3 def
  /SubrCount 5 def

end def
%ADOEndPrivateDict
currentdict end
%ADOEndFontDict
put

dup 1
%ADOBeginFontDict
14 dict begin
<< Font dictionary omitted >>
currentdict end
%ADOEndFontDict
put

dup 2
%ADOBeginFontDict
14 dict begin

```

```

<< Font dictionary omitted >>
  currentdict end
  %ADOEndFontDict
put

def

%%BeginData: 4325480 Binary Bytes
(Binary) 4325452 StartData

<<CIDMap omitted>>
<<SubrMap omitted>>
<<charstrings omitted>>
<<Subroutine Information omitted>>

%%EndData
%%EndResource
%%EOF

```

Comment Conventions

A CIDFont file must begin with the comment characters %!; otherwise it may not be given the appropriate handling in some operating system environments. The first line of the example consists of the following comment:

```
%!PS-Adobe-3.0 Resource-CIDFont
```

The remainder of the line (after the %!), identifies the file as a CIDFont resource that conforms to the PostScript language document structuring conventions version 3.0. Document structuring conventions are explained in the *PostScript Language Reference Manual, Second Edition*.

```
%%DocumentNeededResources: procset CIDInit
%%IncludeResource: procset CIDInit
```

The %%Include construct tells spooler and similar software to determine whether the required resource is available. If the resource is not already available in VM—but is available for downloading—then the spooler should include that resource in-line in the job stream being sent to the interpreter.

The %%BeginResource comment informs spoolers and resource managers that the information which follows is a resource. There is a corresponding %%EndResource comment at the end of the file. The %%BeginResource line also states the type of resource (*CIDFont*) and its name (*Ryumin-Light*).

```
%%BeginResource: CIDFont Ryumin-Light
```

The %%Title comment again states the CIDFont name, and provides the Registry and Ordering strings, and the Supplement number.

```
%%Title: (Ryumin-Light Adobe Japan1 0)
```

The %%Title comment has the following structure:

```
%Title: (<CIDFontName> <registry> <ordering> <supplement>)
```

where *CIDFontName* identifies the CIDFont file, and the remaining fields *<registry>*, *<ordering>*, and *<supplement>* duplicate version control information present elsewhere in the file (primarily as a convenience to parsers). *<registry>* and *<ordering>* are strings that can consist of alphanumerics and the underscore character. No white space is allowed within the string. *<supplement>* is an integer.

The %%Version comment provides the version number of this CIDFont file. This number is an integer; Adobe recommends that it be the same number that is defined for /CIDFontVersion later in the file.

```
%%Version: 1
```

Note The %%Version comment is optional. Adobe encourages its use as an aid to installation software and for future file maintenance.

Additional comments are permitted as long as they conform to the document structuring conventions.

CIDInit Procset Execution Environment

Immediately after the header information and before the definition of the CIDFont proper, a **findresource** is done on the procset *CIDInit*, which is one of the system support files installed on the host or printer hard disk. This ensures that the routines necessary to process CIDFont files are first read into VM. An **end** operator corresponding to this **begin** appears near the end of the file.

```
/CIDInit /ProcSet findresource begin
```

Appendix A contains an explanation of the *CIDInit* procset and *system support files*. Adobe provides these files to developers. See Appendix C for information about how to obtain these and other development files.

CIDFont Resource Dictionary

The line

```
20 dict begin
```

defines and pushes a dictionary onto the dictionary stack. CIDFont is a resource category with an underlying type of *dictionary*; each CIDFont file defines an instance of that category. The **StartData** line near the end of the

example file actually registers the font as a resource instance. Resource categories and their instances are explained in the *PostScript Language Reference Manual, Second Edition*.

Note Because some of the entries described below and in section 4 are optional, the size of dictionary you define may be different from the 20-entry dictionary presented in this example. Level 1 implementations of the PostScript language generate a `dictfull` error if you attempt to define an entry into a dictionary that is already full. No error is generated in Level 2 interpreters. For future extensibility Adobe advises, as was done here, that you define a dictionary containing room for three or four additional entries.

CIDFont Name, Version, and Type

The line beginning with `/CIDFontName` formally defines the name of the CIDFont file. It is the instance name passed to the resource machinery of the PostScript interpreter. Adobe recommends that this be the same name used in the `%%Title` comment.

```
/CIDFontName /Ryumin-Light def
```

The line beginning with `/CIDFontVersion` formally defines the version number of this CIDFont file. If present, this must be the same version number used in the `%%Version` comment.

```
/CIDFontVersion 1 def
```

The line beginning with `/CIDFontType` defines changes to the internal organization of CIDFont files or to the semantics of CIDFont dictionary keys. The `CIDFontType` of the CIDFonts described in this document is 0. The value of `CIDFontType` is an integer.

```
/CIDFontType 0 def
```

The `CIDFontName` and `CIDFontType` are required to be present in the CIDFont file; the `CIDFontVersion` is optional.

Version Control

Version control information is included in the dictionary structure in Example 2::

Example 2: *CIDSystemInfo*

```
/CIDSystemInfo 3 dict dup begin
  /Registry (Adobe) def
  /Ordering (Japan1) def
  /Supplement 0 def
end def
```

This three-element dictionary contains the set of information used for version compatibility checking between CIDFont and CMap files. In addition, each component of the system has its own version field to reflect changes within that component, for example, /CIDFontVersion.

Registry, Ordering, and Supplement entries are required in every CIDFont. There is no length limitation on version control strings (other than the PostScript language limitation of 65535 characters). Version control strings must consist only of alphanumeric characters and the underscore character (_). No white space is permitted.

Registry

Registry is a string value assigned only by the Unique ID coordinator at Adobe Systems. The Registry string identifies an issuer of orderings and is typically a font vendor. For example, the Registry for Adobe Systems is Adobe.

Note See Appendix C for specific information about obtaining Registry strings.

Ordering

The Ordering string uniquely names an ordered character collection within a Registry. For example, an Ordering string within the Adobe Registry is Japan1 and refers to an ordered character collection of 8284 characters.

Different Registries may have identical Ordering strings and operate simultaneously on the same PostScript interpreter because the Registry and Ordering strings, taken together, uniquely identify the character collection.

Supplement

The Supplement integer identifies whether additions have been made to a character collection. The first time a collection is produced by a developer, it should have the Supplement integer 0. As a developer produces incremental additions to that collection, the Supplement number should also be increased by 1 with each release.

Supplement numbers indicate only that additions have been made to the character collection. These additions must follow all previously assigned CID index values. To rearrange or delete characters from a character collection requires defining a new Ordering.

Nonmatching System Information

If the Registry and Ordering strings are identical, a CIDFont and a CMap can be used together. If the Registry and Ordering strings do not match, the two files cannot be used together.

A CMap file and a CIDFont file may have Registry and Ordering strings that match yet have differing Supplement numbers. This may occur if either a CIDFont file or a CMap file has been upgraded, but the other has not.

- When Supplement numbers also match, every mapping in the CMap file results in CIDs that are valid in the CIDFont.
- When the Supplement number in the CMap file is *less* than the Supplement number in the CIDFont file (the CIDFont file is later than the CMap), every mapping in the CMap file results in CIDs that are valid in the CIDFont. However, the CIDFont will have extra CIDs available that cannot be produced by the earlier CMap file.
- When the Supplement number in the CMap file is *greater* than the Supplement number in the CIDFont file (the CIDFont file is earlier than the CMap), some mappings from the later CMap file result in CIDs that are not valid in the CIDFont file. CID 0, the default notdef character, is used in this event.

FontBBox

FontBBox is a required key that defines in an arbitrary space of 1000/em a box large enough to enclose any of the characters in the CIDFont.

Every glyph in the character collection corresponds to one or another of the font dictionaries in the FDArray, and each of the font dictionaries has a FontMatrix key. That FontMatrix key controls the character space for all characters using that font dictionary. Typically, the FontMatrix is 1000 units to the em—but not necessarily so. Because FontMatrix may not always be 1000 units to the em, FontBBox is defined in an arbitrary space that does consist of 1000 units to the em. See the *PostScript Language Reference Manual, Second Edition* or *Adobe Type 1 Font Format* for an explanation of FontBBox.

```
/FontBBox [-180 -293 1090 1010] def
```

Unique Identification Numbers

The CIDFont file defines two types of unique ID numbers. Unique ID numbers are necessary so that fonts can be cached between jobs. The first type of unique ID has a UIDBase value in the CIDFont file and a UIDOffset value in the CMap file. The second type has an XUID (*extended unique ID*) number in the CIDFont file only. The XUID number is a Level 2 feature; it is ignored by Level 1 interpreters. Unique IDs are explained in more detail in section “Calculating Unique IDs in Appendix D.

The first type (UIDBase + UIDOffset) is intended for Level 1 interpreters with composite font extensions or for Level 2 interpreters that do not offer *native mode* support for CID-keyed fonts (as defined in Section 2). The XUID

method is intended for Level 2 interpreters that can offer native mode support. Adobe recommends using *both* types of unique ID numbers for backward compatibility as well as for continued future compatibility. Both types of unique ID numbers are optional.

Unique ID Type: UIDBase

The line

```
/UIDBase 27611 def
```

sets the starting or *base* value for a group of unique ID numbers for the CIDFont. Each CMap file has an entry that gives the *offset* from this base for its particular character set and encoding. When a CID-keyed font is created in VM, the base and offset values are used to create unique ID numbers “on the fly” as required. Both parts work together to ensure that there is no collision between an ID assigned to a CIDFont and an ID assigned to any other font program.

Note UIDBase numbers are assigned by Adobe Systems. UIDOffset numbers are calculated by the font developer. The typical maximum count of consecutive numbers available for a CIDFont is 2,000; larger and smaller ranges are available on request.

Unique ID Type: XUID

An XUID (*extended unique ID*) is an entry whose value is an array of integers. This array identifies a font by the entire sequence of numbers in the array.

The line

```
/XUID [1 11 27611] def
```

defines an XUID array. The XUID array in the CIDFont file has no relationship to the XUID in the CMap file.

The first element of an XUID array must be a unique *organization identifier*, assigned by Adobe Systems. Appendix C explains how to obtain such an identifier. In the example, the value 1 identifies the organization as Adobe Systems. The remaining elements, and the allowed length of XUIDs starting with that organization ID, are the responsibility of the organization to which the organization ID has been assigned. An organization can establish its own registry for managing the space of numbers in the second and subsequent elements of XUID arrays.

The organization ID value 1000000 is reserved for private interchange in closed environments. XUID arrays starting with that number may be of any length.

FontInfo

The FontInfo dictionary is optional and contains information for PostScript language programs using the CIDFont resource, or as human-readable documentation. The *PostScript Language Reference Manual, Second Edition* describes the various FontInfo keywords that are valid and how they are used by application programs.

Example 3: FontInfo dictionary

```
/FontInfo 2 dict dup begin
  /Notice ((c) Copyright 1993 Adobe Systems Incorporated. All
  Rights Reserved.) def
  /FullName (Ryumin-Light) def
end def
```

Accessing Charstring Data

As stated before, there are two parts to the CIDFont file: a PostScript language program, and a data section. The data section can contain four blocks:

- a CIDMap that associates a font dictionary index with a glyph descriptor value used to access charstring data with each CID,
- one or more SubrMaps that associate a descriptor used to access subroutine data with each subr index,
- the subroutines used by the charstring data, and
- the charstrings that contain glyph descriptions.

This section describes the format of the CIDMap and how it is used to access charstring data.

CIDMap Format

Example 4: provides information necessary to access and interpret the CIDMap.

Example 4: CIDMap offset

```
/CIDMapOffset 0 def
/FDBytes 1 def
/GDBytes 3 def
/CIDCount 8284 def
```

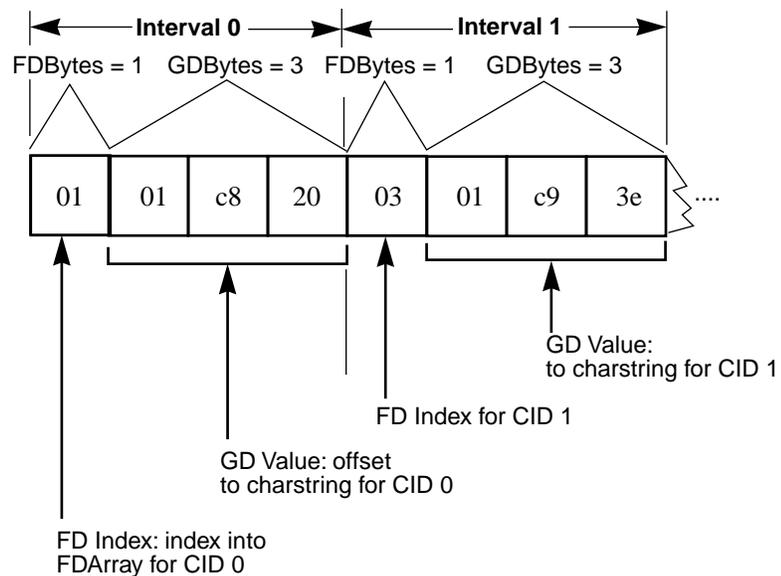
The CIDMapOffset is the byte location relative from the start of the data section of the CIDFont file. See the section “Defining the CIDFont Resource and the Data Section,” for a more precise definition of the start of the data section.

The keywords `FDBytes` and `GDBytes` have values corresponding to the number of bytes used to store the *font dictionary* (FD) index and the *glyph descriptor* (GD) value, respectively, for each CID in the CIDMap. The sum of these two byte lengths is the length of one interval in the CIDMap, and is used in conjunction with a CID to determine how many bytes from the beginning of the CIDMap to locate the interval containing the data for that CID.

If `FDBytes` is equal to 0, the CIDMap contains no FD indices, and the FD index of 0 is assumed.

The GD value is an offset relative from the start of the data section to the desired charstring. Figure 3 shows how these intervals are organized.

Figure 3 *Internal organization of the CIDMapOffset string*



Note All Japanese language fonts Adobe has produced to date use one byte to index into the FDArray and three bytes of offset information per character description. Your values may differ.

Because the length of a charstring for a given CID is defined as the difference between its GD value and the value of the successor GD, charstrings must be contiguous and in increasing order. As a consequence of this, it is possible to omit glyphs for CIDs from a CIDFont by making their GD value and successor GD values the same. An interval for a CID having this property is called an *empty interval*.

Also note that to compute the length of the last charstring, an extra interval is needed which follows the interval for the last CID. This interval is called the *last interval*. The GD value for the last interval must be one more than the final byte of the charstring for the last CID. The FD index for the last interval is undefined if FDBytes is greater than 0.

The first CIDMap interval, which is indexed by CID 0, contains the FD index and GD value for the default notdef character. All CIDFonts must include a default notdef character—the appearance of the glyph assigned to CID 0 or pointed to by CID 0 in each CIDFont, as with other glyphs, is left to the font designer. Section 5 discusses in detail the circumstances in which the CMap resource instance decodes character codes to the character ID of 0.

The keyword CIDCount defines how many CIDs are defined in the character collection. A CIDCount of n indicates CIDs from 0 to $n - 1$, and a CIDMap will have $n + 1$ intervals, including the last interval.

Building Subset CIDFonts

It is sometimes especially useful to build a CIDFont containing a subset of all the glyphs for its character collection. Such a font is called a *subset font*. For example, a font vendor might want to build a Kana subset of a full Japanese language font. Or, a developer might want to omit certain infrequently used glyphs. Glyph data might not be available for some characters in a character collection; still it might be desirable to build such a CIDFont.

In these cases, an empty interval is used to indicate that glyph data is missing. For example, in Figure 4, the second font is missing the “B” glyph.

Figure 4 *Empty intervals*

<pre>/FDBytes 1 def /FDArray 2 dict begin ... end</pre>	<pre>/FDBytes 0 def /FDArray 1 dict begin ... end</pre>																																			
<table><thead><tr><th>CID/ Interval</th><th>FD</th><th>GD</th><th>Character</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>100</td><td>"A"</td></tr><tr><td>1</td><td>1</td><td>200</td><td>"B"</td></tr><tr><td>2</td><td>0</td><td>350</td><td>"C"</td></tr><tr><td>3</td><td>0</td><td>400</td><td>--</td></tr></tbody></table>	CID/ Interval	FD	GD	Character	0	0	100	"A"	1	1	200	"B"	2	0	350	"C"	3	0	400	--	<table><thead><tr><th>CID/ Interval</th><th>GD</th><th>Character</th></tr></thead><tbody><tr><td>0</td><td>100</td><td>"A"</td></tr><tr><td>1</td><td>200</td><td>--</td></tr><tr><td>2</td><td>200</td><td>"C"</td></tr><tr><td>3</td><td>250</td><td>--</td></tr></tbody></table> <p>Length = 0 Empty interval implied</p>	CID/ Interval	GD	Character	0	100	"A"	1	200	--	2	200	"C"	3	250	--
CID/ Interval	FD	GD	Character																																	
0	0	100	"A"																																	
1	1	200	"B"																																	
2	0	350	"C"																																	
3	0	400	--																																	
CID/ Interval	GD	Character																																		
0	100	"A"																																		
1	200	--																																		
2	200	"C"																																		
3	250	--																																		

In the figure, the “full” font on the left has a character collection of three characters, and has a CIDMap that has intervals corresponding to each character, and one additional (last) interval. It also has two font dictionaries in its FDArray.

The “subset” font also has a character collection of three characters, though the glyph for “B” is not present. This font also has a CIDMap that has intervals corresponding to each character, and an additional (last) interval. Because only one font dictionary was needed in the subset font, the optimization of setting FDBytes to 0 was used. Notice that although the GD values for both intervals 1 and 2 are the same, the computed lengths for the charstring data indicate that interval 1 is an empty interval (the length of the charstring equals 0), while interval 2 has glyph data (the length of the charstring equals 50).

If the CID references an empty interval, the appropriate notdef character will be selected instead.

FDArray: Overall Structure

The FDArray is an array of font dictionaries. A font dictionary contains essential hinting information that is used, along with a charstring, to render a glyph. An entry in each font dictionary that stores this information is another dictionary called Private. Given the large collection of characters possible in a CIDFont, it is likely that there will be groups of glyphs that are similar and which can be hinted alike. Such groups reference the same font dictionary.

Note Although the font dictionaries in the FDArray contain most of the essential entries of a well-formed font dictionary (as defined in Adobe Type 1 Font Format), these are not font dictionaries on which to do a **findfont**, **definefont**, or other such operations.

The example following shows the overall structure of an FDArray and omits individual font dictionary content which was shown at the beginning of section 3.2, “CIDFont Example,.” This array contains three font dictionaries, but another CIDFont may have more or fewer according to the number of hint groups needed.

Example 5: FDArray

```

/FDArray 3 array

dup 0
  %ADOBEGINFontDict
  14 dict begin
  << Font dictionary omitted >>
  currentdict end
  %ADOEndFontDict
put

dup 1
  %ADOBEGINFontDict
  14 dict begin
  << Font dictionary omitted >>
  currentdict end
  %ADOEndFontDict
put

dup 2
  %ADOBEGINFontDict
  14 dict begin
  << Font dictionary omitted >>
  currentdict end
  %ADOEndFontDict
put

def

```

Every charstring must reference one of the font dictionaries defined in this array, and every CIDFont must have an FDArray with at least one font dictionary.

Each font dictionary in a CIDFont of CIDFontType 0 is a font dictionary as described in the *PostScript Language Reference Manual, Second Edition*, with certain exceptions. These font dictionaries may be Type 1 or Type 3 font dictionaries, but must not include the following entries:

Type 1 Exceptions

Encoding Array Should not be present in an FDArray font dictionary because the CMap file controls encoding.

Charstring Dictionary Should not be present in an FDArray font dictionary because charstring information appears in a data block near the end of the CIDFont file.

Subrs Array Should not be present in an FDArray font dictionary because subroutine information appears in a data block near the end of a CIDFont file along with charstrings and offset and index information.

Type 3 Exceptions

Encoding Array Should not be present in an FDArray font dictionary because the CMap file controls encoding.

Handling Subroutine Information

The information that is handled by the Type 1 Subrs array must be organized differently in a CIDFont. In Type 1 font programs, Subrs subroutines for charstrings are defined in the Private dictionary, but they are stored in the data section of CIDFonts. OtherSubr subroutines are defined in the Private dictionary of CIDFonts.

Within the Private dictionary of the example are defined three keywords with values such as in Example 6::

Example 6: *Three keywords in the Private dictionary*

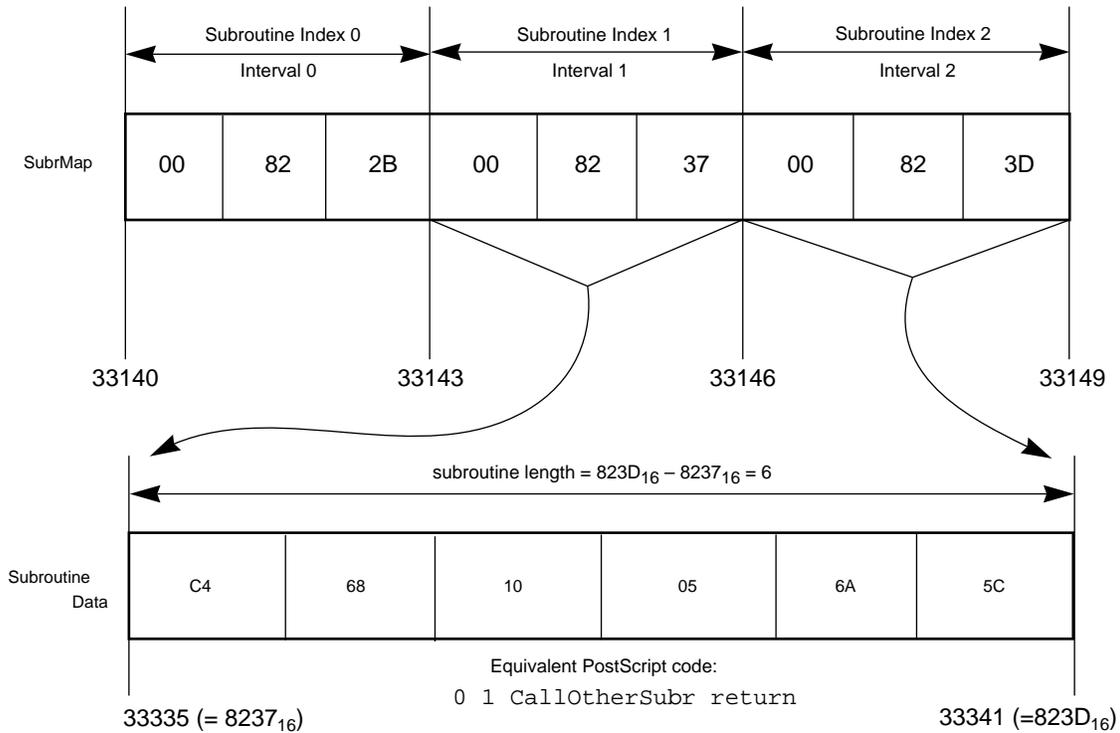
```
/SubrMapOffset 33140 def
/SDBytes 3 def
/SubrCount 5 def
```

The SubrMapOffset is the byte offset relative from the start of the data section of the CIDFont to the beginning of the SubrMap, a sequence of intervals containing *Subroutine Descriptor* (SD) values used to access subroutine data. SD values are typically offsets to subroutine data, but in some data organizations may be indices.

The keyword SDBytes defines the number of bytes needed to store the SD value, and is the length of one interval in the SubrMap. If these three entries are not present in the Private dictionary, there are no subroutines.

The SubrMapOffset, SDBytes, and subroutine index determine how many bytes from the beginning of the SubrMap are needed to locate the interval containing the SD value for that subroutine index. The length of a subroutine for a given subroutine index is defined as the difference between its SD value and that of the successor SD value; therefore, a last interval for SubrMaps is needed, just as with the CIDMap. Figure 5 shows how the SubrMap relates to the length of subroutine data.

Figure 5 Relationship of SubrMap to subroutine data length



Because the subroutine information appears in a font dictionary, and because there can be more than one font dictionary in the FDArray, it follows that there can be more than one SubrMap. If there is more than one SubrMap, Adobe recommends that they be organized contiguously; the value of SubrMapOffset in each font dictionary points to the start of the SubrMap for that particular font dictionary. There is only one Subroutine data section, the subroutine information within it organized contiguously.

Defining the CIDFont Resource and the Data Section

Having all components defined for the CIDFont resource, it is necessary to register that resource, signal the end of the PostScript language program, and begin the data section. This is accomplished with the **StartData** procedure, as in Example 7.

The comment **%%BeginData** and its corresponding **%%EndData** bracket the data section of the file for parsers, spoolers, and ATM-J. See the *PostScript Language Reference Manual, Second Edition* for more specific information about the **%%BeginData** comment.

The data following the **StartData** procedure name includes the CIDMap, the SubrMaps, the subroutine data, and the charstring data (typically in that

order) and begins one byte following the procedure name. If the first argument to **StartData** is Binary, then this byte *must* be a space character (0x20). If the first argument is Hex, then any white space characters may be used.

Example 7: The StartData procedure

```
%%BeginData: 4325480 Binary Bytes
(Binary) 4325452 StartData
<<Data begins one space following StartData>>
<<CIDMap omitted>>
<<SubrMap omitted>>
<<Subroutine Information omitted>>
<<charstrings omitted>>
%%EndData
%%EndResource
```

Note The **StartData** procedure that comes “stock” with the compatibility mode *CIDInit* procset is designed for file-based CIDFonts. If you need to load a CIDFont into VM, Adobe will provide a different version of **StartData**.

The **StartData** procedure is defined by the *CIDInit* procset. **StartData** registers the CIDFont resource. */CIDFontName* is the key associated with this instance.

The comment %%EndResource ends the file.

4 CIDFont Reference

This section summarizes information presented in section 3 and provides additional information on topics not covered there. Primarily, it documents information about each keyword in the PostScript language portion of a CIDFont file. The detailed explanation is presented in alphabetical order by keyword name.

4.1 CIDFont Organization

A CIDFont has two parts: a PostScript language program section and a data section. The PostScript portion produces a CIDFont resource instance, which is a dictionary object, and defines a variety of keys. The data section contains charstrings, their subroutines, and data used to access them.

Keyword Organization

This section provides a list of CIDFont dictionary keys (with the exception of **CIDInit**, and **StartData**, which are procedure names); some keys are optional. In a file, each key takes a value, and must be properly defined as a member of a dictionary. See the sample file in Section 3 for an example of constructing a CIDFont resource using these keys.

CDevProc	<i>(optional)</i>
CIDInit	<i>(required) Procedure name</i>
CIDFontName	<i>(required)</i>
CIDFontVersion	<i>(optional)</i>
CIDFontType	<i>(required)</i>
CIDSystemInfo	<i>(required)</i>
FontBBox	<i>(required)</i>
UIDBase	<i>(optional)</i>
XUID	<i>(optional)</i>
FontInfo	<i>(optional)</i>
CIDMapOffset	<i>(required)</i>
FDBytes	<i>(required)</i>
GDBytes	<i>(required)</i>
CIDCount	<i>(required)</i>
FDArray	<i>(required)</i>
StartData	<i>(required) Procedure name</i>

The first part of the file (up to but not including the data section) is a self-contained PostScript language program. It ends with **StartData**, and produces a CIDFont resource instance in VM. The data section is not placed in VM and remains on disk.

Data Section

The data section can contain four items:

- A CIDMap, which contains information about the location of each charstring in the CIDFont and the font dictionary that corresponds to it.
- One or more SubrMaps, which contain information about the location of each subroutine used by the characters in the font. SubrMaps are optional, depending on whether the font dictionaries in the FDArray require subroutines.
- The subroutines used by the glyph descriptions. Subroutines are optional, depending on whether the font dictionaries in the FDArray require them.
- The charstrings, which contain the glyph descriptions.

4.2 CIDFont Resource Keys

This section summarizes in alphabetical order the keys that are understood in a CIDFont resource dictionary. The type of each key (for example, *integer*) appears after its name, along with whether that key is required in the CIDFont file.

CDevproc procedure optional

The **CDevProc** procedure algorithmically derives global changes to a font's metrics. See the *PostScript Language Reference Manual, Second Edition* for more extensive information about using **CDevProc** in font programs.

CIDCount integer required

The **CIDCount** key provides the number of valid character IDs in the CIDFont. Valid CIDs are in the range of 0 to **CIDCount** – 1, inclusive.

CIDFontName name required

This keyword sets the name of the CIDFont resource instance. That name is the key subsequently used to identify this resource instance. It is very important for **CIDFontName** to conform to the naming conventions for CIDFonts. Naming conventions are discussed in Appendix D.

CIDFontType integer required

The **CIDFontType** keyword tells what is in the font resource, how it is organized, and how it is represented. All CIDFonts described in this document have a **CIDFontType** of 0. Other **CIDFontType** values are reserved.

CIDFontVersion integer optional

The **CIDFontVersion** formally defines the version number of this CIDFont file. This should be the same version number used in the %%Version comment.

CIDMapOffset integer required

The **CIDMapOffset** is the byte offset of the CIDMap relative from the start of the data section of the CIDFont file. See section 4.3, “Defining the CIDFont Resource,” for a more precise definition of the start of the data section.

CIDSystemInfo dictionary required

The **CIDSystemInfo** dictionary is required. It is important in maintaining version control among the component files that make up the CID-keyed font. The string keywords to this dictionary have the standard PostScript language limit of 65535 bytes; however they may contain only alphanumeric characters and the underscore (_) character—white space is not permitted. See section 3, “CIDFont Tutorial,” for examples of how to use the **CIDSystemInfo** keywords.

The **CIDSystemInfo** dictionary must contain the following three keywords.

Registry string required

Registry is a string value assigned by the Unique ID Coordinator at Adobe Systems. An example of the Registry keyword and value is:

```
/Registry (Adobe) def
```

Ordering string required

The Ordering string uniquely identifies the ordered glyph collection of the CIDFont within its Registry. Two different Registry values may have the same Ordering string. An organization is responsible for maintaining its own set of Ordering strings. An example of the Ordering keyword and value is:

```
/Ordering (Japan1) def
```

Supplement integer required

The Supplement integer identifies any additions to the glyph collection of a CIDFont. Such additions must not alter the existing ordering of the collection (in which case, the Ordering string would change).

FDArray array required

The FDArray is an array of font dictionaries. A font dictionary contains essential hinting information which is used, along with a charstring, to render a glyph.

Every charstring must reference one of the font dictionaries defined in this array, and every CIDFont must have an FDArray with at least one font dictionary.

The value for FDBytes determines how many bytes are used as an index into the FDArray and, hence, the range of font dictionaries that can be referenced. For example, With an FDBytes value of 1, a CIDFont's FDArray can have up to 256 referenced font dictionaries (numbered 0 to 255).

eexec encryption is not required for CIDFontType 0 fonts.

See section 3 for a complete discussion of how the font dictionaries in the FDArray access subroutines.

FDBytes integer required

FDBytes has a value corresponding to the number of bytes used to store the *font dictionary* (FD) index for each CID in the CIDMap. If FDBytes is equal to 0, the CIDMap contains no FD indices, and the FD index of 0 is assumed.

FontBBox array required

FontBBox is a required key that defines in an arbitrary space of 1000/em a box large enough to enclose any of the characters in the CIDFont. See the *PostScript Language Reference Manual, Second Edition* or *Adobe Type 1 Font Format* for an explanation of FontBBox.

FontInfo dictionary optional

This keyword holds the font name, weight, and any copyright notice. See *PostScript Language Reference Manual, Second Edition*, and *Adobe Type 1 Font Format* for more information about the FontInfo dictionary keyword.

GDBytes integer required

GDBytes has a value corresponding to the number of bytes used to store the glyph descriptor (GD) value for each CID in the CIDMap. The GD value is an offset relative from the start of the data section to the desired charstring.

UIDBase integer optional

UIDBase complements an entry in the CMap file (UIDOffset). Together, their data make up a two-part system based on both the CIDFont and the CMap files for assigning unique IDs in VM. See section 5 for an explanation of how both values work together.

UIDBase is a number in the range 0 to 16,777,215 (or $2^{24} - 1$), and is assigned by Adobe Systems. See Appendix C for specific information about obtaining UIDBase numbers from Adobe Systems.

Note: UIDBase (and UIDOffset) are useful only in compatibility mode. Adobe suggests including them for backwards compatibility.

XUID integer optional

An XUID (extended unique ID) is an entry whose value is an array of integers. This array identifies a font by the entire sequence of numbers in the array. For example, the line

```
/XUID [1 11 27611] def
```

defines an XUID array. The XUID array in the CIDFont file has no relationship to the XUID in the CMap file.

Note: XUID is useful only in native mode. Adobe strongly suggests including an XUID to help ensure future compatibility.

4.3 Defining the CIDFont Resource

The **StartData** procedure registers the CIDFont resource, proceduralizes how the data section of the CIDFont file is handled by the PostScript interpreter, and signals the beginning of the data section of the CIDFont. The data section consists of the CIDMap, charstrings, any SubrMaps, and any Subrs. The **StartData** procedure is defined in the **CIDInit** procset.

The syntax of **StartData** is

```
(<string> <int> StartData
```

where the value of *<string>* can be Binary or Hex to specify how the data is encoded, and the value of *<int>* is the number of bytes of data *after decoding*. This data must begin one byte after the **StartData** procedure call is encountered in the data stream or file. If the first argument to **StartData** is Binary, then this byte *must* be a space character (0x20).

If **StartData** is executed when using a CIDFont from a file-based system, it

- defines the CIDFont resource,
- removes the CIDFont instance from the dictionary stack,
- executes a **currentfile closefile**,
- removes the CIDInit procset instance from the dictionary stack.

If **StartData** is executed when a CIDFont is to be loaded into VM, it

- creates one data object in the CIDFont resource dictionary to hold the data. This object is made up of one or more PostScript language string objects, depending on the size of the data.
- creates a second object in the CIDFont resource dictionary to act as a CIDMap analog. The GD value in this object is an index, rather than an offset.
- defines the CIDFont resource.
- removes the CIDFont instance from the dictionary stack.
- removes the CIDInit procset instance from the dictionary stack.

StartData allows data to be organized as binary or as ASCII hexadecimal values. ASCII hexadecimal is useful for transmitting data when using binary might cause problems. Loading a CIDFont onto an file-based system, however, must result in the data section of a file being organized in a binary format, even if the data is transmitted as ASCII hexadecimal.

Data encoded as ASCII hexadecimal is converted to binary as follows. For each pair of ASCII hexadecimal digits (0-9 and A-F or a-f), one byte of binary data is produced. All white space characters—tab, carriage return, linefeed, formfeed, and null—are ignored. The character > indicates end of data (EOD); if the data section is ASCII hexadecimal, it must end with this end-of-data character. Any other characters cause an **ioerror**. If the decoding filter encounters EOD when it has read an odd number of hexadecimal digits, it behaves as if it has read an additional zero digit.

Here are two examples of using the **StartData** procedure.

Example 8 Using the *StartData* procedure

```
%%BeginData: 2484 Binary Bytes
(Binary) 2460 StartData
<<2460 binary bytes of data omitted>>
%%EndData

%%BeginData: 4942 Binary Bytes
(Hex) 2460 StartData
<<2460 pairs of ASCII hex data omitted>>
<<+ 1 EOD marker>>
%%EndData
```

The `%%BeginData` comment states the number of binary bytes in the data section, plus (in this case) 24 and 22 additional bytes. The difference between the comment value and the value used in the procedure call is the number of characters in the procedure call line itself (plus one). This is so parsers and spoolers can have an accurate character count based on the location of the comment, and so the actual byte count of the data (which starts after the procedure call) can be accurate, too.

In the ASCII example, there are 2460 pairs of values, for a total of 4920 bytes. The offset for the call (21) plus the EOD marker (1) make for the difference as shown.

Of course, the number of additional bytes in any particular situation may be different from this example, depending on whether the **StartData** procedure takes the (Binary) or (Hex) string argument, and on the number of characters that make up the integer argument.

5 CMap Tutorial

A CMap file defines the relationship between a *character code* and the *character description* delivered by the CIDFont program to the rasterizer.

The specific set of characters to which a CMap refers is called a *character set* or *charset*. Various CMap files specifying different charsets can refer to the same CIDFont; similarly, the same CMap file can refer to various fonts. The

mapping of input code to character ID defines the *encoding* imposed on the charset. A CMap file is an ASCII text file; its format is a subset and extension of the PostScript language, with its own syntactical rules.

It is unlikely that a font developer will need to build a CMap file for Japanese language fonts. Adobe Systems makes available CMap files for the most common charset and encoding combinations, as defined by Japanese national standards groups. However, a developer will need to build a CMap file when creating a font for a charset or encoding not provided by Adobe.

This tutorial covers the mapping of character codes to CIDs for a single CIDFont. CMaps are more general than this; they can also map to codes or names in a base font, and they can map a single space of codes into character selectors for multiple fonts and CIDFonts. However, compatibility mode restricts a CMap to a single CIDFont. As native mode devices become more available, additional documentation will describe the extensions necessary to support it.

5.1 CMap File Components

A CMap file specifies the character descriptions to which an input code maps. The character may be identified by a character ID, a character name, or a character code. The file contains header comments, information for ensuring compatibility with CIDFont files, caching identification data, the writing mode, a definition of *codespace* (the set of valid input codes), and code mapping information.

When executed, a CMap file creates a PostScript language resource instance of type CMap in VM. The resource is implemented as a dictionary. See the *PostScript Language Reference Manual, Second Edition* for more information about resource instances and their types.

Two examples of a CMap file follow in this section. Each is complete. The first is “stand-alone,” in that it does not use information from any other CMap file. The second example incorporates information from another CMap file in order to make its own definition of the input codes and the corresponding glyphs smaller.

5.2 First Example: Stand-Alone CMap File

This example is a full and complete CMap file that does not use information from any other CMap files. Where something has been omitted, there is explanatory text between brackets, << *like this* >>.

Example 9 Stand-alone CMap file

```
%!PS-Adobe-3.0 Resource-CMap
%%DocumentNeededResources: procset CIDInit
```

```

%%IncludeResource: procset CIDInit
%%BeginResource: CMap 83pv-RKSJ-H
%%Title: (83pv-RKSJ-H Adobe Japan1 0)
%%Version: 1

/CIDInit /ProcSet findresource begin

12 dict begin

begincmap

/CIDSystemInfo 3 dict dup begin
  /Registry (Adobe) def
  /Ordering (Japan1) def
  /Supplement 0 def
end def

/CMapName /83pv-RKSJ-H def

/CMapVersion 1 def
/CMapType 0 def

/UIDOffset 0 def
/XUID [1 10 25324] def

/WMode 0 def

4 begincodespacerange
  <00> <80>
  <8140> <9ffc>
  <a0> <df>
  <e040> <fbfc>
endcodespacerange

1 beginnotdefrange
<00> <1f> 1
endnotdefrange

100 begincidrange
<20> <7e>1
<8140> <817e> 633
<8180> <81ac> 696
<81b8> <81bf> 741
<81c8> <81ce> 749

<< 90 ranges missing >>

<9540> <957e> 3475
<9580> <95fc> 3538
<9640> <967e> 3663
<9680> <96fc> 3726
<9740> <977e> 3851
endcidrange

100 begincidrange
<9780> <97fc> 3914
<9840> <9872> 4039
<989f> <98fc> 4090

```

```
<9940> <997e> 4184
<9980> <99fc> 4247
```

```
<< 90 ranges missing >>
```

```
<ed83> <ed83> 7934
<ed84> <ed84> 992
<ed85> <ed85> 7935
<ed86> <ed86> 994
<ed87> <ed87> 7936
endcidrange
```

```
17 begincidrange
<ed88> <ed8d> 996
<ed8e> <ed8e> 7937
```

```
<< 13 ranges missing >>
```

```
<ee9a> <ee9a> 768
<ee9b> <ee9c> 7631
endcidrange
```

```
endcmap
```

```
CMapName currentdict /CMap defineresource pop
```

```
end
end
%%EndResource
%%EOF
```

Comment Conventions

A CMap file must begin with the comment characters %!; otherwise it may not be given the appropriate handling in some operating system environments. The first line in the file is

```
%!PS-Adobe-3.0 Resource-CMap
```

The remainder of the line (after the %!) identifies that file as a CMap resource that conforms to the PostScript language document structuring conventions version 3.0. Document structuring conventions are explained in the *PostScript Language Reference Manual, Second Edition*.

In VM, the CMap uses a procset from a *system support file* named *CIDInit*. Appendix A explains about system support and other files that may be required by a particular PostScript interpreter. For the benefit of parsers and spoolers, a CMap file carries the header lines

```
%%DocumentNeededResources: procset CIDInit
%%IncludeResource: procset CIDInit
```

%%DocumentNeededResources indicates that an external resource is needed by this document; in this case, the procset **CIDInit**. %%IncludeResource tells any handling software that if the resource is not available on the PostScript interpreter, it should be included in-line if possible.

The %%BeginResource comment informs spoolers and resource managers that the information that follows is a resource. There is a corresponding %%EndResource comment at the end of the file. The %%BeginResource line also states the type of resource (*CMap*) and its name (*83pv-RKSJ-H*).

```
%%BeginResource: CMap 83pv-RKSJ-H
```

The %%Title comment again states the CMap name, and provides the Registry and Ordering strings, and the Supplement number.

```
%%Title: (83pv-RKSJ-H Adobe Japan1 0)
```

The %%Title comment has the following structure:

```
%Title: (<CMapName> <registry> <ordering> <supplement>)
```

where *CMapName* identifies the CMap file, and the remaining fields *registry*, *ordering*, and *supplement* duplicate version control information present elsewhere in the file (primarily as a convenience to parsers). The variables *registry* and *ordering* are strings that can consist of alphanumerics and the underscore character. No white space is allowed within the string. The variable *supplement* is an integer.

The %%Version comment provides the version number of this CMap file. Adobe recommends that it be the same number that is defined for CMapVersion later in the file.

```
%%Version: 1
```

Additional comments are permitted as long as they conform to the document structuring conventions.

Initializing the CID Procset

Immediately after the header information and before the definition of the CMap proper, a **findresource** operation is run on the file *CIDInit*, which is one of the system support files installed in the file system. This ensures that the routines necessary to process CMap files are first read into VM. An **end** operator corresponding to this **begin** appears near the end of the file.

```
/CIDInit /ProcSet findresource begin
```

Appendix A contains an explanation of the *CIDInit* (and other) system support files.

CMap Resource Dictionary

After the CID procset has been initialized, the file defines a PostScript language resource instance whose underlying type is a dictionary. The line

```
12 dict begin
```

begins this dictionary. The line that uses the operator **defineresource** near the end of the file registers the CMap as a resource instance.

Note To accommodate structures that are built in VM, Adobe recommends that you allocate five more elements to this dictionary than those that appear to be directly consumed by the code. Using fewer elements than this may result in a dictfull error on Level 1 interpreters. No such error occurs on Level 2 interpreters.

Establishing the CMap

After the CMap resource dictionary has been established, the definition of the CMap can take place. The process adds several key-value pairs to the CMap resource dictionary that are not apparent from the PostScript language code in the CMap file, and which explain the extra dictionary elements in the preceding line.

The CMap is begun with the line

```
begincmap
```

There is a corresponding **endcmap** operator near the end of the file that completes the task of building the resource.

Version Control

The first of the dictionary objects is CIDSystemInfo which contains the version control information. CID-keyed fonts implement version control to ensure compatibility between this CMap file and the CIDFont files used with it.

CIDSystemInfo can be defined as either a dictionary or an array of dictionaries. When CIDSystemInfo is defined as a dictionary, the dictionary must contain three key-value pairs that compose the version control information; Registry, Ordering, and Supplement. The values of Registry and Ordering are character strings. Supplement has an integer value.

The following is used in the example of an stand-alone CMap file, showing CIDSystemInfo being defined as a dictionary.

Example 10 CIDSystemInfo as dictionary

```
/CIDSystemInfo 3 dict dup begin
```

```

    /Registry (Adobe) def
    /Ordering (Japan1) def
    /Supplement 0 def
end def

```

When CIDSytemInfo is defined as an array of dictionaries, each dictionary in the array must contain three entries described above. The example below shows CIDSytemInfo being defined as an array containing two dictionaries.

Example 11: CIDSytemInfo as an array of dictionaries

```

/CIDSytemInfo 2 array dup
[
  0 3 dict begin
    /Registry (Adobe) def
    /Ordering (Japan1) def
    /Supplement 0 def
  end def put

  1 3 dict begin
    /Registry (Adobe) def
    /Ordering (Japan1) def
    /Supplement 1 def
  end def put
]
def

```

It is important that the Registry and Ordering strings of the CMap file match those of the CIDFont file with which it works. “Version Control” in the section 2 provides an explanation of how these values are obtained. That section includes a discussion of what can happen when the Supplement values of a CMap file and a CIDFont don’t match.

CMap Name, Version, and Type

The line beginning with CMapName formally defines the name of the CMap file. It is the instance name passed to the resource machinery of the PostScript interpreter. Adobe strongly recommends that this be the same name used in the %%Title comment.

```

/CMapName /83pv-RKSJ-H def

```

The line beginning with CMapVersion formally defines the version number of this CIDFont file. If present, this must be the same version number used in the %%Version comment.

```

/CMapVersion 1 def

```

The line beginning with CMapType defines changes to the internal organization of CMap files or the semantics of CMap operators. The CMapType of CMaps described in this document is 0. The value of CMapType is an integer.

```

/CMapType 0 def

```

The CMapName and CMapType are required to be present in the CMap file; the CMapVersion is optional.

Unique Identification Numbers

The CMap file contains two types of unique ID numbers. Unique ID numbers are necessary so that caching can take place between jobs. The first type of unique ID uses the UIDOffset value in the CMap file and a corresponding UIDBase value in the CIDFont file. This process is explained in more detail in Appendix D. The second method uses an XUID (extended unique ID) number which is not related to a similar number in the CIDFont file. The XUID number is a Level 2 feature; it is ignored by Level 1 interpreters.

Unique ID Type: UIDOffset

The line

```
/UIDOffset 0 def
```

sets the *offset* of unique ID numbers for the character set described by this file. Each CMap file must have its own set of unique ID numbers different from those of other CMap files that reference the same character collection. See section 3 for information about UIDBase.

Note UIDBase numbers are assigned by Adobe Systems. UIDOffset numbers are calculated by the font developer. The typical maximum count of consecutive numbers available for a CIDFont is 2,000; larger and smaller ranges are available on request.

Unique ID Type: XUID

An XUID (extended unique ID) is an entry whose value is an array of integers. This array identifies a font by the entire sequence of numbers in the array. The line

```
/XUID [1 10 25324] def
```

defines an XUID array.

The first element of an XUID array must be a unique *organization identifier*, assigned by Adobe Systems. Appendix C explains how to obtain such an identifier. Section 3 discusses XUID numbers for CIDFont files; that information is also valid here.

Writing Mode

The `WMode` dictionary entry controls whether the CID-keyed font writes horizontally or vertically. It indicates which set of metrics will be used when a base font is shown. An entry of 0 defines horizontal writing from left to right; an entry of 1 defines vertical writing from top to bottom. Other values for `WMode` are reserved.

```
/WMode 0 def
```

`WMode` in the `CMap` overrides any `WMode` in any font or CIDFont referred to by the `CMap` file.

Codespace

The `CMap` file fully describes the potential set of valid input character code values. Input codes may consist of one, two, three, or more hexadecimal bytes, expressed between `< >` brackets. Ranges need not be contiguous, but cannot overlap. The codespace definition unambiguously specifies which input codes consist of one byte, which consist of two, and so forth. The definition of codespace *must* precede any code mappings, including any `not-defs`—this is one of the few strict organizational requirements of the `CMap` file.

The following example shows the definition of codespace for the first example:

Example 12 Codespace

```
4 begincodespaceraange
  <00> <80>
  <8140> <9ffc>
  <a0> <de>
  <e040> <fbec>
endcodespaceraange
```

The line

```
4 begincodespaceraange
```

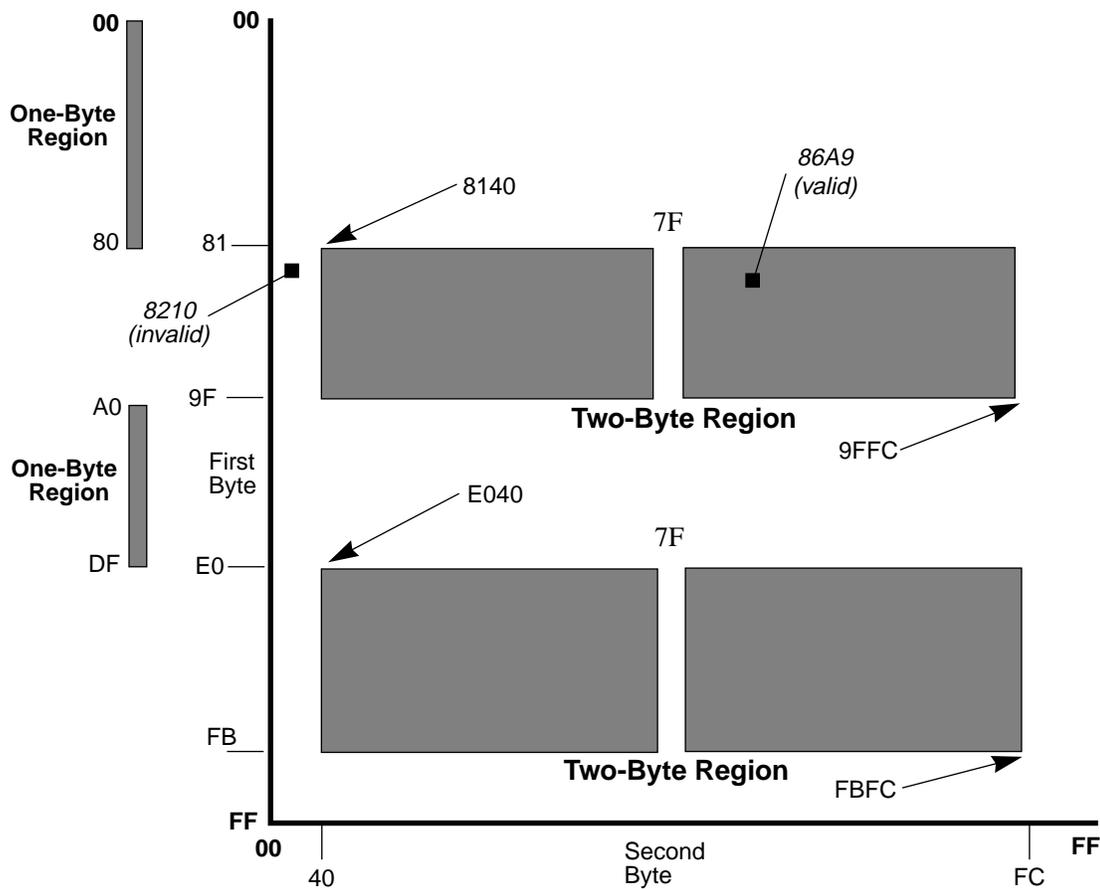
defines four codespace entries. The codespace entries themselves consist of pairs of hexadecimal numbers in the form `<low-end> <high-end>`.

A set of codespace ranges can have up to and including 100 definition lines. This (and other similar limitations) helps avoid stack overflow errors on earlier interpreters. If a `CMap` requires more than 100 lines to define its codespace ranges, it can use several sets of 100 or fewer.

Codespace is not necessarily *linear*; the number of bytes required to express the limits of the codespace range also indicates the dimensionality of that range. Figure 6 shows how the codespace definition in this example comprises two single-byte *linear* ranges of codes (<00> to <80> and <A0> to <DF>) and two double-byte *rectangular* ranges of codes (<8140> to <9FFC> and <E040> to <FBFC>). The first two-byte region comprises *all* codes bounded by first-byte values of 81 through 9F and second-byte values of 40 through FC. Thus, the input code <86A9> is *within* the region because both bytes are within bounds. That code is *valid*. The input code <8210> is *not* within the region, even though its first byte is between 81 and 9F, because its second byte is not within bounds. That code is *invalid*. The second two-byte region is similarly bounded.

Note Overlapping codespaces are not permitted.

Figure 6 Codespace ranges for the 83pv-RKSJ-H charset encoding



In this example, the codespace range from <00> to <80> consists of single-byte codes. In the Japanese language font *Ryumin-Light-83pv-RKSJ-H*, these are proportionally spaced Roman characters. The codespace <8140> to

<9FFC> consists of full-width Kanji characters. The range <A0> to <DF> contains half-width Kana, and the range <E040> to <FBFC> contains another set of full-width Kanji.

Code Mappings

A CMap file maps input codes within the codespace to a *character selector* and *component font index* that actually accesses the glyph. The component font index identifies the specific font, and the character selector identifies the character within that font that is to be displayed. A character selector can be a character ID for a CIDFont, a character code, or a glyph name—the latter two are for accessing Type 1 and Type 3 fonts that may be part of a CID-keyed font. For most purposes, this combination of character selector and component font index is transparent, and it is useful to think of them as one item.

As shown in Example 13, the *cidrange* sections associate the beginning and ending of a range of acceptable character codes, expressed as hexadecimal strings, with the starting CID for that range. Code mappings can also associate input codes with character codes or glyph names, if needed.

Example 13 Code mappings

```
100 begincidrange
    <20>    <7e>    1
    <8140>  <817e>  633
    <8180>  <81ac>  696
    <81b8>  <81bf>  741
    <81c8>  <81ce>  749

<< 90 ranges missing >>

    <9540>  <957e>  3475
    <9580>  <95fc>  3538
    <9640>  <967e>  3663
    <9680>  <96fc>  3726
    <9740>  <977e>  3851
endcidrange

100 begincidrange
    <9780>  <97fc>  3914
    <9840>  <9872>  4039
    <989f>  <98fc>  4090
    <9940>  <997e>  4184
    <9980>  <99fc>  4247

<< 90 ranges missing >>

    <ed83>  <ed83>  7934
    <ed84>  <ed84>  992
    <ed85>  <ed85>  7935
    <ed86>  <ed86>  994
    <ed87>  <ed87>  7936
endcidrange
```

```

17 begincidrange
   <ed88> <ed8d> 996
   <ed8e> <ed8e> 7937

<< 13 ranges missing >>

   <ee9a> <ee9a> 768
   <ee9b> <ee9c> 7631
endcidrange

```

As with codespace ranges, there can be up to 100 code mapping ranges in each set. When more than 100 are required, the CMap uses several sets. The first line of each mapping states how many sets of input codes and starting CIDs there are in the range—in the case of this example, a total of 217 in three ranges of 100, 100, and 17. Succeeding lines within each range state a specific starting input code, a specific ending input code, and the starting CID for that range. The starting and ending input codes appear as hexadecimal strings expressed within `<>` brackets; the CID is a decimal number with no brackets.

There are 94 input codes between `<20>` and `<7E>`. Because the starting CID is (decimal) 1, input code `<20>` corresponds to character ID 1, `<21>` corresponds to 2, `<22>` corresponds to 3, and so forth. Input code `<7E>` corresponds to character ID 94.

There are three important requirements of code mappings:

- Code mappings (unlike codespace ranges) may overlap, but succeeding maps supercede preceding maps.
- The domain of the code mappings must lie entirely within the codespace.
- The domain of the code mappings may be multidimensional if the codespace is multidimensional.

The operator **endcidrange** finishes code mapping for ranges of input.

Notdef Ranges

Input codes may be presented to the CMap resource instance that do not map to valid character IDs according to the information in the codespace and code mapping definitions. These are handled by showing notdef characters. The default notdef character is always accessed by CID 0. Every CID-keyed font must have a default notdef character. However, a developer can assign valid input codes to the default notdef character and to notdef characters other than the default.

- As shown in Figure 6, an input code that falls outside of valid codespace is *invalid*. When an input code is presented to the CMap resource instance that does not map to a valid codespace, the default notdef character will be substituted and shown.
- If the Supplement numbers do not match between CIDFont and CMap resources, an input code may be presented that does not map to an existing character; in this case the default notdef character will be substituted and shown.
- If the input code is for an *empty interval* (as explained in section 3), the notdef character may be the default or one assigned by the developer, depending on the notdef mapping.
- A developer can also explicitly assign a notdef to one or more valid input codes.

Note The name `.notdef` is the glyph name of a character required to be present in Type 1 and Type 3 fonts. The term “notdef,” as used in the context of this document, is a generic name Adobe uses to describe a glyph that will be shown if some encoding does not result in a showable combination of component font index and glyph selector.

The example shows how a developer can map valid input codes to specific notdef characters.

```
1 beginnotdefrange
  <00> <1f>1
endnotdefrange
```

The first line states how many ranges of notdef definitions there are—in the case of this example, there is one. As with codespace ranges and code mappings, up to and including 100 notdef ranges can be specified in each set, with several sets of 100 or fewer permitted.

The two hexadecimal strings (`<00>` and `<1f>`) state the bounds of the range of input codes. The decimal number states the single character ID to which all codes in that range are mapped *if* a notdef must be shown. For example, if a character ID falling within the notdef range is presented to the CMap resource instance, which for some reason (such as an empty interval) cannot show a glyph, then the notdef character defined here will be shown instead. Developer-defined notdefs such as this operate *only* when a CID that falls in range cannot otherwise produce a glyph; they can thus coexist with ranges of valid mappings.

Note Notdef characters are selected from the same collection as all other characters. The character corresponding to character ID 1 is a notdef character—and happens also to be the first character in the code mapping range <00> to <1f>.

The ability to specify several notdef characters is useful for fonts such as those of the Japanese language, where there are several character subsets of various widths. Adobe CMap files, for example, include half-width kana, full-width Kanji, and proportional roman characters. Each subset has one notdef character of its own, specifically the half-width space, that full-width space, and the proportional space.

5.3 Closing the CMap File and Creating the Resource Instance

The last five lines of the CMap file explicitly end the CMap information, establish the CMap resource, and formally close the file.

Example 14 The end of the CMap file

```
endcmap

CMapName currentdict /CMap defineresource pop

end
end
%%EndResource
%%EOF
```

The operator **endcmap** corresponds to the operator **begincmap** that appears at the beginning of the file. The two operators bracket the CMap information.

The line

```
CMapName currentdict /CMap defineresource pop
```

explicitly states the encoding for this CMap file, defines it as a VM resource, and pops it from the stack. The argument **CMapName** is the *instance key*, defined earlier in the file. The argument **CMap** is the *resource category*. See Appendix D for important information about CMap naming conventions.

The two **end** operators correspond (respectively) to the **dict begin** line and the **CIDInit** procset invocation.

The comment

```
%%EndResource
```

is a comment that defines the end of the file in accordance with the document structuring conventions. It is useful if this CMap file is concatenated with other files in a job stream.

The comment

```
%%EOF
```

formally signals the end of the file.

5.4 Second Example: A CMap File That Uses Another

One CMap resource instance can use the VM structures already created by another instance. This second example (Example 15) shows how this is done. Most of the 8000-plus Kanji characters are the same whether written horizontally or vertically; a few are different. This example shows a complete CMap file for a vertical Japanese font that uses the characters already mapped for a horizontal font, and which then goes on expressly to map only those characters that are different.

Example 15 *CMap file that uses another CMap file*

```
!PS-Adobe-3.0 Resource-CMap
%%DocumentNeededResources: procset CIDInit
%%DocumentNeededResources: CMap Ext-RKSJ-H
%%IncludeResource: procset CIDInit
%%IncludeResource: CMap Ext-RKSJ-H
%%BeginResource: CMap Ext-RKSJ-V
%%Title: (Ext-RKSJ-V Adobe Japan1 0)
%%Version: 1

/CIDInit /ProcSet findresource begin

12 dict begin

begincmap

/CIDSystemInfo 3 dict dup begin
  /Registry (Adobe) def
  /Ordering (Japan1) def
  /Supplement 0 def
end def

/Ext-RKSJ-H usecmap

/CMapName /Ext-RKSJ-V def
/CMapVersion 1 def
/CMapType 0 def

/UIDOffset 800 def
/XUID [1 10 25316] def

/WMode 1 def

1 begincidrange
<8141> <8142> 7887
endcidrange

35 begincidchar
```

```

<8143> 8286
<8144> 8274
<814a> 8272
<8387> 7936
<838e> 7937

<< 30 ranges missing >>

endcidchar

endcmap

CMapName currentdict /CMap defineresource pop

end
end
%%EndResource
%%EOF

```

The header for a CMap file that uses another is the same as that for a stand-alone CMap file, with the addition of a `%%DocumentNeededResources` and an `%%IncludedResources` comment referring to the CMap being *used*. A 12-element dictionary is also established and the **begincmap** operator is issued.

The important operator in Example 15 is **usecmap**. It appears in the line

```
/Ext-RKSJ-H usecmap
```

You can express the same resource instances in VM without using this operator (by duplicating the contents of the other file in line), but some implementations can make more efficient use of CMap resources when one file uses another than if each file were to be defined separately.

The **usemap** operator allows one resource instance to *refer* to the VM structures already created by another. The amount of VM saved is related to the relative sizes of the files. If one file creates a structure with 217 CID ranges (comprising over 8200 characters), and another file can use them by remapping only 37 characters, as in Example 15, VM savings can be substantial. The **usemap** operator must appear before any range operation.

After the line with the **usecmap** operator are lines for defining CMapName (note the `-V` to denote the vertical orientation of *this* CMap file), CMapVersion, UIDOffset, and XUID—all following the same syntax and usage as with a stand-alone CMap file.

The WMode entry gives the writing mode of the *using* file. The using file adopts the codespace, character mappings, and notdefs of the CMap being used unless they are specifically redefined. It causes an error to try to redefine the adopted codespace.

Example 15 redefines a single two-character range of input codes using the **begincidrange** and **endcidrange** operators and 35 individual characters using the **begincidchar** and **endcidchar** operators.

The resource instance is created by Example 16:

Example 16 *Creating the resource instance*

```
endcmap

CMapName currentdict /CMap defineresource pop

end
end
%%EndResource
%%EOF
```

which state the instance key and the resource category, as do the similar lines in the first example.

5.5 CMap File Naming Convention

While it is expected that the CMap files provided by Adobe Systems will be adequate for the majority of CID-keyed fonts, some font developers will need to develop their own CMap files. To ensure that CMap file names are unique, Adobe Systems recommends that the file name use the developer's Registry name as a prefix to the CMap file name.

It is recommended that the Registry name that is registered with Adobe Systems be limited to two or three characters. It is important to keep it short since the CMap file name will appear in the font menu, where there may be system-imposed length constraints. The Registry name should consist of only alphabetic characters and numbers. For example, a font developer who has registered the string *ABC* might name their custom CMap file: *ABC-H*

Note Since the names of CMap files developed by Adobe Systems are expected to be useful to and shared by other Asian font developers, applications software developers should not expect to see Adobe's Registry name as a prefix for CMap file names.

6 Rearranged Font Tutorial

Because they have many characters, Japanese fonts can occupy several megabytes of disk space. Often, a developer will want to produce a set of similar fonts, each font differing from others by such details as the style of proportional Roman characters, the weight of Kana, or the inclusion of special gaiji characters not available in the original font.

A *rearranged font* can produce the effect of multiple versions of the same original (or *template*) font, but without the storage overhead of an extra ten or twenty megabytes. It produces this effect by “borrowing” characters from other fonts. Rearranged fonts are small in size; Adobe has found that they typically occupy fewer than 30 kilobytes each.

Rearranged fonts can make use of CID-keyed fonts, existing composite fonts (also called Japanese Type 1 fonts), Roman Type 1 fonts, and Type 3 fonts. A software developer can create a rearranged font from an existing font without being concerned with the format of the font programs that make up the rearranged font. A developer will need to know the character set and encoding of the font programs from which characters are being borrowed.

This section describes how to produce rearranged fonts, from the standpoint of the developer who wishes to produce a collection of fonts as variations on a single template font. After reading it, a developer should be able to construct a rearranged font that incorporates glyphs from several existing fonts.

6.1 Rearranged Font Components

A rearranged font consists of a CMap file that uses two special commands: **beginrearrangedfont** and **endrearrangedfont**. The rearranged font file uses a slightly different header from a CMap file, and uses an additional complement of CMap operators to accomplish the rearrangement.

In the rearranged font are named a *template font* and one or more *component fonts*. The template font provides the structure on which the rearranged font is built, and the component fonts provide the borrowed characters.

Rearranged fonts themselves contain no character data. They *describe* the fonts from which the template font is to borrow certain characters, and how those characters are to be mapped to input codes within the rearranged font’s codespace. A rearranged font is thus a *recipe* for creating a new font. The effect of executing a rearranged font is to create a composite font in VM. The rearranged font behaves in just the same way as any other CIDFont: the name of the rearranged font appears on font menus, the font can be downloaded to a printer, and it can be used with ATM-J.

There are two major restrictions on the use of rearranged fonts.

- Although a rearranged font file uses CMap operators, its mapping is specific to the template font being rearranged; it does not have the “generalized” nature of a CMap file (which may be used with many different CID-keyed fonts).
- All component fonts of a rearranged font must be available to the PostScript interpreter at **findfont** time.

6.2 Rearranged Font Example

This section presents an example of a rearranged font. Where statements or data have been omitted, they are replaced with explanatory text within brackets, like this:

<< text here omitted >>

Like a CMap file, a rearranged font file is a program written in the PostScript language. The order and syntax of entries is important; Section 7 describes them in detail.

The example shows several of the rearrangements a developer might want to make to an existing Japanese language font. It starts with a font named *Jun101-Light-83pv-RKSJ-H*. The *Jun101-Light* “starting” font is referred to as the *template font*, because the rearrangements are built on it.

Example 17 is a rearranged font; it makes the following four changes:

1. It replaces the single-byte proportional Roman characters of *Jun101-Light-83pv-RKSJ-H* with characters from the Type 1 font *Poetica-ChanceryIV*.
2. It replaces the punctuation characters of *Jun101-Light-83pv-RKSJ-H* with the punctuation characters of *ShinseiKai-CBSK1-83pv-RKSJ-H*.
3. It replaces the Hiragana and Katakana characters of *Jun101-Light-83pv-RKSJ-H* with characters from *FutoGoB101-Bold-83pv-RKSJ-H* and *FutoMinA101-Bold-83pv-RKSJ-H*.
4. It adds one row of gaiji from the Type 1 font *HSMInW3Gai30*.

The template font defines the codespace of a rearranged font. Codespace is explained in section 5. Characters from all component fonts must *conform* to the input codespace of the template font. For example, if the codespace of the template font has no valid codes assigned between <8100> and <81FF>, then the input code <8121> (which may be valid in a JIS-encoded font) will be interpreted as <81> <21> in the rearranged font. As is shown later in the example, input codes for the borrowed characters from a component font must be mapped to input codes that are valid for the template font.

Example 17 A rearranged font

```
%!PS-Adobe-3.0 Resource-Font
%%ADOResourceSubCategory: RearrangedFont
%%DocumentNeededResources: procset CIDInit
%%+ font Jun101-Light-83pv-RKSJ-H
%%+ font Poetica-ChanceryIV
%%+ font ShinseiKai-CBSK1-83pv-RKSJ-H
%%+ font FutoGoB101-Bold-83pv-RKSJ-H
%%+ font FutoMinA101-Bold-83pv-RKSJ-H
```

```

%%+ font HSMInW3Gai30
%%IncludeResource: procset CIDInit
%%IncludeResource: font Poetica-ChanceryIV
%%IncludeResource: font HSMInW3Gai30
%%BeginResource:Font Jun101-Light-K-G-R-83pv-RKSJ-H
%%Version: 1

/CIDInit /ProcSet findresource begin

%%ADOSTartRearrangedFont
/Jun101-Light-K-G-R-83pv-RKSJ-H

[ /Jun101-Light-83pv-RKSJ-H
  /Poetica-ChanceryIV
  /ShinseiKai-CBSK1-83pv-RKSJ-H
  /FutoGoB101-Bold-83pv-RKSJ-H
  /FutoMinA101-Bold-83pv-RKSJ-H
  /HSMInW3Gai30
] beginrearrangedfont

% substitute Roman characters with JIS reencoding
1 beginusematrix [1 0 0 1 0 0.15] endusematrix
1 usefont

5 beginbfchar
<27> /quotesingle
<5c> /yen
<60> /grave
<7e> /tilde
<7f> <7f>
endbfchar

4 beginbfrange
<00> <26> <00>
<< 2 ranges omitted >>
<61> <7d> <61>
endbfrange

% substitute punctuation
2 usefont

8 beginbfchar
<815c> <815c>
<< 6 ranges omitted >>
<eb63> <eb63>
endbfchar

14 beginbfrange
<8141> <8147> <8141>
<< 12 ranges omitted >>
<eb8c> <eb8d> <eb8c>
endbfrange

% substitute hiragana
3 usefont
20 beginbfrange

```

```

<8152> <8153> <8152>
<< 18 ranges omitted >>
<ed80> <ed96> <ed80>
endbfrange

% substitute katakana
4 usefont
14 beginbfrange
<8154> <8155> <8154>
<< 12 ranges omitted >>
<ec9f> <ecf1> <ec9f>
endbfrange

% substitute single row of gaiji characters
5 usefont
1 beginbfrange
<f000> <f0ff> 0
endbfrange

endrearrangedfont
end
%%EndResource
%%EOF

```

Comment Conventions

A rearranged font resource file must begin with the comment characters %!; otherwise it may not be handled correctly in some operating system environments. The first two lines in the file are

```

%!PS-Adobe-3.0 Resource-Font
%%ADOResourceSubCategory: RearrangedFont

```

The remainder of the first line (after the %!) identifies that file as a rearranged font resource that conforms to the PostScript language document structuring conventions version 3.0. Document structuring conventions are explained in *PostScript Language Reference Manual, Second Edition*.

The following comment lines state that the CIDInit procset is required, and lists the set of Japanese fonts from which characters are borrowed.

Example 18 Fonts used in the rearranged font

```

%%DocumentNeededResources: procset CIDInit
%%+ font Jun101-Light-83pv-RKSJ-H
%%+ font Poetica-ChanceryIV
%%+ font ShinseiKai-CBSK1-83pv-RKSJ-H
%%+ font FutoGoB101-Bold-83pv-RKSJ-H
%%+ font FutoMinA101-Bold-83pv-RKSJ-H
%%+ font HSMInW3Gai30

```

The `%%Include` construct in the lines following tells spooler and similar software to determine whether the required resource is available. If the resource is not already available in VM—but is available for downloading—then the spooler should include that resource in-line in the job stream being sent to the interpreter.

```
%%IncludeResource: procset CIDInit
%%IncludeResource: font Poetica-ChanceryIV
%%IncludeResource: font HSMInW3Gai30
```

The `%%BeginResource` comment informs spoolers and resource managers that the information that follows is a resource. There is a corresponding `%%EndResource` comment at the end of the file. The `%%BeginResource` line also states the type of resource (*RearrangedFont*) and its name (*Jun101-Light-K-G-R-83pv-RKSJ-H*). Suggestions for how to name fonts appear in Appendix D.

```
%%BeginResource: Font Jun101-Light-K-G-R-83pv-RKSJ-H
```

The `%%Version` comment provides the version number of this CMap file.

```
%%Version: 1
```

Additional comments are permitted as long as they conform to the document structuring conventions.

Initializing the CID Procset

Immediately after the header information and before the definition of the rearranged font, a **findresource** is executed on the file *CIDInit*, which is one of the system support files installed on the host or printer hard disk. This ensures that the routines necessary to process the rearranged font file are present in VM. An **end** operator corresponding to this **begin** appears at the end of the file.

```
/CIDInit /ProcSet findresource begin
```

Appendix A contains an explanation of the *CIDInit* (and other) system support files.

Component Fonts

The fonts that comprise a rearranged font are called *component fonts*. The **beginrearrangedfont** operator defines which fonts become component fonts and states the name of the resultant rearranged font. There is a corresponding **endrearrangedfont** operator near the end of the file.

The **beginrearrangedfont** operator takes two operands: a name object that is the name of the rearranged font and a *component fonts array* that is a list of component fonts, portions of which comprise the rearranged font. All component fonts for a rearranged font must be present on a PostScript interpreter when the font is executed. In the example, the **beginrearrangedfont** statement looks like this:

Example 19 *Component fonts of the rearranged font*

```
%ADOSTartRearrangedFont
/Jun101-Light-K-G-R-83pv-RKSJ-H

[ /Jun101-Light-83pv-RKSJ-H
  /Poetica-ChanceryIV
  /ShinseiKai-CBSK1-83pv-RKSJ-H
  /FutoGoBl01-Bold-83pv-RKSJ-H
  /FutoMinA101-Bold-83pv-RKSJ-H
  /HSMInW3Gai30
] beginrearrangedfont
```

The first line, `/Jun101-Light-K-G-R-83pv-RKSJ-H`, is the name of the rearranged font that results from executing this file. See Appendix D for suggestions about font naming, which is very important to the proper execution of CID-keyed fonts.

The array operand begins with the name of the template font. All rearrangements are performed on a logical copy of this font. Succeeding elements of the array are font names, each of which contain characters that will be borrowed for the specific rearrangements described.

Because this operand of **beginrearrangedfont** is an array, each component font can be referred to by its position in the array, with the template font considered to be *font 0*. The **usefont** operator, and the **beginusematrix** and **endusematrix** operators (which appear several lines later in the file), refer to the fonts in this array by number. The **beginrearrangedfont** component fonts array must be specified before any **usefont**, **beginusematrix**, or **endusematrix** operator is used.

Note If you wish to use the same component font in two (or more) different ways, it must appear in the **beginrearrangedfont** array more than once. For example, if you wish to use a component font both with and without a transformation matrix, that font must appear twice in the array. This is because the **usefont**, **beginusematrix**, or **endusematrix** operators identify a component font by its position in the array, and all instances of the font at that position are modified accordingly. To use a component font both with and without a matrix, therefore, requires two separate instances of that font in the array.

In this example:

- The name of the resulting rearranged font is *Jun101-Light-K-G-R-83pv-RKSJ-H*.
- The template font is *Jun101-Light-83pv-RKSJ-H*.
- Proportional Roman characters are borrowed from *Poetica-ChanceryIV*.
- Punctuation characters are borrowed from *ShinseiKai-CBSK1-83pv-RKSJ-H*.
- Hiragana characters are borrowed from *FutoGoB101-Bold-83pv-RKSJ-H*.
- Katakana characters are borrowed from *FutoMinA101-Bold-83pv-RKSJ-H*.
- New gaiji characters are borrowed from *HSMInW3Gai30*.

Replacing and Adjusting Roman Characters

The rearranged font in this example specifies that the Roman characters in the template font should be replaced by characters from the *Poetica-ChanceryIV* Type 1 font, and that the borrowed Roman characters should be adjusted by changing their baseline. The default value for the Roman character baseline in a Japanese font is 120/1000 em. This value might be inappropriate for the substitute Roman characters in a particular font. You may also wish to change the baseline for the default Roman characters. In the example, the baseline used to position the characters in relation to the Japanese characters is to be raised by 150/1000 em. The following code performs this adjustment:

```
1 beginusematrix [1 0 0 1 0 0.15] endusematrix
```

The first argument is the index of the font in the component fonts array to which the matrix adjustment should be applied (in this case, the *first* element in the array, *Poetica-ChanceryIV*). *Every* character borrowed from that particular component font will use the transformation specified by the matrix.

The effect of the **beginusematrix** and **endusematrix** operators is equivalent to applying **makefont** to the base font, using the same matrix. That is, the resulting FontMatrix is the result of concatenating the font's original FontMatrix with the matrix specified by **beginusematrix** and **endusematrix**, in that order. (Matrix multiplication is not commutative.)

The **beginusematrix** and **endusematrix** operators also can be used with characters from a Japanese language font, for example, to achieve rotation, or artificial skewing or obliquing. The matrix commands are not restricted to proportional Roman characters; this example, however, uses them that way.

Because the template font is shift JIS-encoded, the Type 1 proportional Roman characters also will be shift JIS-encoded. When a Roman font is shift JIS-encoded, a small number of characters differ from the standard ASCII encoding—the backslash becomes the yen sign, and the shift JIS tilde is used in place of the ASCII tilde.

The **usefont** operator specifies the font in the component fonts array from which characters are borrowed; in this case, the *first* element in the array, *Poetica-ChanceryIV*.

```
1 usefont
```

All operators in the file that follow, until any succeeding **usefont**, now borrow characters only from *font 1* in the component fonts array—*Poetica-ChanceryIV*.

Note Because of the many **begin** and **end** forms of CMap operators, this document uses a tilde ~ to denote that the **begin** and the **end** prefixes are left off. For example, both the **beginusematrix** and the **endusematrix** operators can be referred to together as the **~usematrix** operators. This form is not used when only one or the other operator is meant.

A **usefont** must appear before any **~bfchar** or **~bfrange** operator is specified.

The code mappings that follow the **usefont** explicitly identify by input code the Roman characters that are to be substituted in the template font, and the individual characters—and ranges of characters—from the component font that are to be used to replace them. All input codes must be valid in the codespace of the template font. Input can be single codes or ranges of codes, and the outputs can be character codes or names.

The first five single character substitutions implement the shift JIS reencoding. For example the input code <5c> in the template font is made to correspond to the yen symbol and so on. The **beginbfchar** and **endbfchar** operators bracket one or more single characters being drawn from a Type 1 or Type 3 *base font* (hence the **bf**). The first element on each line is the input code of the template font; the second element is the code or name of the character in the Type 1 font that will correspond to that code in the rearranged font.

Example 20 *Base font characters used in the rearranged font*

```
5 beginbfchar
<27> /quotesingle
<5c> /yen
<60> /grave
<7e> /tilde
<7f> <7f>
endbfchar
```

The remaining four range substitutions complete the Roman character substitution. The first and second elements in each line are the beginning and ending valid input codes for the template font; the third element is the beginning character code for the range of proportional Roman characters being assigned to that template input range.

Example 21 *Ranges of base fonts*

```
4 beginbfrange
<00> <26> <00>
<28> <5b> <28>
<5d> <5f> <5d>
<61> <7d> <61>
endbfrange
```

Note Roman characters placed in a Japanese font that has the 83pv-RKSJ-H character set and encoding fall in the range of <00> to <7f>.

Replacing Punctuation Characters

After the proportional Roman characters have been added to the rearranged font, the operator

```
2 usefont
```

signifies that operations are now to be performed on *font 2* of the component fonts array—in this case, *ShinseiKai-CBSK1-83pv-RKSJ-H*.

The code mappings which in Example 22 explicitly identify the punctuation characters to be substituted in the template font, and the individual characters and ranges of characters from the component font which will be used to replace them.

Example 22 *Substituting punctuation characters*

```
8 beginbfchar
<815c> <815c>
<< 6 ranges omitted >>
<eb63> <eb63>
endbfchar

14 beginbfrange
<8141> <8147> <8141>
<< 12 ranges omitted >>
<eb8c> <eb8d> <eb8c>
endbfrange
```

Replacing Hiragana and Katakana Characters

The next code fragment from the example substitutes the Hiragana characters in the template font with the *same* Hiragana characters from another font. Typically, this type of rearrangement substitutes characters of a different *style*—say, Kana Large or Kana Old Style—from those that are already included in the template font.

The **usefont** operator indicates that rearrangements are to be drawn from *font 3* of the component fonts array, */FutoGoB101-Bold-83pv-RKSJ-H*.

```
3 usefont
```

The code mappings that follow explicitly identify the ranges of Hiragana characters to be replaced in the template font and the corresponding ranges in the component font from which characters are to be borrowed.

Example 23 Replacing Hiragana

```
20 beginbfrange
<8152> <8153> <8152>
<< 18 ranges omitted >>
<ed80> <ed96> <ed80>
endbfrange
```

The **usefont** operator again points to *font 4* in the component fonts array, */FutoMinA101-Bold-83pv-RKSJ-H*. This time, the rearranged font file substitutes the full set of Katakana characters in the template font.

The code mappings that follow explicitly identify the ranges of Katakana characters to be replaced in the template font and the corresponding ranges in the component font from which characters are to be borrowed.

Example 24 Replacing Katakana

```
4 usefont
14 beginbfrange
<8154> <8155> <8154>
<< 12 ranges omitted >>
<ec9f> <ecf1> <ec9f>
endbfrange
```

Adding Gaiji Characters

Most character set and encoding combinations used in Japan today reserve a number of rows for gaiji characters. The size and portion of the code space available for such gaiji characters varies with character set and encoding combination. The Apple® Macintosh® *83pv-RKSJ* combination, for example, reserves 12 rows (<F0>-<FB>) for gaiji characters. Each row can contain up to 188 characters.

The **usefont** operator selects *font 5* in the component font array, in this case /*HSMInW3Gai30*, a Type 1 font that contains a selection of gaiji characters.

```
5 usefont
```

The code mappings that follow explicitly identify the ranges of UserGaiji space into which to substitute the gaiji characters borrowed from that component font.

```
1 beginbfrange
<f000> <f0ff> 0
endbfrange
```

The range <f000> to <f0ff> provides 256 character positions—the same number of character positions available in a typical Roman font. The most effective way to add gaiji is to “drop in” the 256-character gaiji font into the UserGaiji row. Thus, character 32 in the gaiji font in this situation appears at position <f020>.

Building the Font

The **endrearrangedfont** operator ends the rearranged font information and defines the new Japanese font in VM by performing the rearrangements as described, and registering the resulting Japanese font for system use.

The comment %%EndResource conforms to the document structuring conventions.

The operator **end** concludes the procset initiated after the header.

```
endrearrangedfont
end
%%EndResource
%%EOF
```

7 CMap Reference

This section contains detailed information about the operators that can be used in a CMap file. It is divided into four parts.

First, there is a discussion of file nomenclature and lexical elements. Second, there is a summary of operators, organized into groups of related functions. The summary is intended to help locate the operators needed to perform specific tasks. Third, there is a section that describes the organizational requirements of a CMap file. Fourth, there is a detailed description of all operators, organized alphabetically by operator name. Because of the many begin/end pairs of operators, operators are listed alphabetically under their *begin* versions.

Note The word operators as used in this section refers to a set of executable commands that are defined in the CIDInit procset resource. While they may look like and have syntax similar to PostScript language operators, they are not part of the PostScript language.

Each operator is presented in the following format:

operator *operand*₁ *operand*₂...*operand*_n **operator** *result*₁...*result*_m

A detailed explanation of the operator appears here.

Example

An example of the use of the operator appears here. The symbol \Rightarrow designates the values (if any) left on the operand stack by the example.

Errors

A list of possible errors that this operator might execute appears here. Please note, however, that there are font interpreters that are not PostScript interpreters; ATM-J, for example, is not a full PostScript interpreter. Only when executing a CMap file containing errors on a PostScript interpreter will the file produce predictable error behavior.

See Also

A list of related operators may appear here.

At the head of an operator description, *operand*₁ through *operand*_n are the operands that the operator requires, with *operand*_n being the topmost operand on the stack. The operator pops these objects from the stack and consumes them. After executing, the operator leaves the objects *result*₁ through *result*_m on the stack, with *result*_m being the topmost element.

7.1 CMap File Nomenclature and Lexical Elements

Section 5 discussed CMap files, which control the codespace and encoding of a CID-keyed font. Section 6 explained rearranged fonts, which use CMap commands and borrow characters from various CID-keyed fonts to create a new font.

As used in Operator Summary and Operator Details, the nature of the *source* and *destination* argument differs depending on whether the CMap commands are acting as part of a rearranged font. These source and destination codes also can be different in length within a single operation.

srcCode

In CMap files, *srcCode* refers to the input codes that are to be mapped into a variety of character selectors: *dstCodes*, *dstCharNames*, or *dstCIDs*. The `<xxxx>` hexadecimal string notation specifies single- or multiple-byte input codes, where each pair of hexadecimal digits represents a byte of the code.

When CMap commands are used as part of a rearranged font, *srcCode(s)* refer to the character codes in the *template font* that will be replaced in the rearranged font with characters borrowed from one of the component fonts.

dstCode or dstCharName

In CMap files, *dstCodes*, *dstCharNames*, or *dstCIDs* represent the selector that will be used to extract a glyph from a font resource. Table 1 shows how various selectors access a glyph.

Table 1 *Relationship of input code to selector*

<i>Selector</i>	<i>Font Type</i>
CID (integer)	CIDFont
Single-byte code (hex string)	Type 1 font program
Name (name object)	Type 1 font program
Single-byte code (hex string)	Type 3 font program
Name (name object)	Type 3 font program
Single-byte code (hex string)	Type 0 font program
Double-byte code (hex string)	Type 0 font program

When CMap commands are being used as part of a rearranged font, the *dstCIDs*, *dstCodes*, or *dstCharNames* specify those characters from the component font that are to be selected and shared by the rearranged font. Table 2 shows the lexical elements that are supported in CMap files and rearranged fonts.

Table 2 *PostScript language lexical elements*

<i>Representation</i>	<i>Meaning</i>
<code>%...</code>	comments
<code>nnn</code>	integer and real numbers
<code>/abc</code>	literal name objects
<code>(abc)</code>	string objects
<code><xxxx></code>	hexadecimal string notation

[...]	array syntax
operator	only the operators described in this section may be used

7.2 Operator Summary

Operators in CMap files fall into five groups, based on usage.

General Operators

/CMapName	usecmap –	uses another CMap’s VM resource
fontID	usefont –	specifies font used subsequently
fontID	beginusematrix –	
[a b c d tx ty]	endusematrix –	transformation matrix to use with font
int	begincodespacerange –	
srcCode ₁ srcCode ₂	endcodespacerange –	sets valid input codes
–	begincmap –	
–	endcmap –	brackets CMap definition in file

Operators That Use Character Names or Character Codes as Selectors

int	beginbfchar –	
srcCode dstCode	<i>or</i>	
srcCode dstCharname	endbfchar –	specifies one base font glyph
int	beginbfrange –	
srcCodeLo srcCodeHi dstCodeLo	<i>or</i>	
srcCodeLo srcCodeHi		
[/dstCharname ₁ .../dstCharname _n]	endbfrange –	specifies range of base font glyphs

Operators That Use CIDs as Selectors

int	begincidchar –	
srcCode dstCID	endcidchar –	specifies one CIDFont character
int	begincidrange –	
srcCodeLo srcCodeHi dstCIDLo	endcidrange –	specifies range of CIDFont characters

notdef Operators

int	beginnotdefchar –	
srcCode dstCID	endnotdefchar –	specifies one notdef character
int	beginnotdefrange –	
srcCodeLo srcCodeHi dstCIDLo	endnotdefrange –	specifies range of notdef characters

Rearranged Font Operators

While not actually CMap operators, rearranged font operators are listed here for completeness.

/newFontName [component fonts array]	beginrearrangedfont –	identifies fonts used in rearrangement
--------------------------------------	------------------------------	--

– **endrearrangedfont** – font built on an existing template

7.3 CMap File Overview

Several parts of a CMap file must appear in a particular order. This section provides that organizational information and a brief explanation of why the ordering must take place. A CMap file must comply with the following rules:

1. Header comments must appear first. In particular, the first line of the file must be constructed as explained in sections 5 and 6.
2. The CIDInit procset **findresource** call appears immediately after the header information.
3. The **begincmap** operator must appear before any range operators. It and the **endcmap** operator (see below) bracket the entire CMap dictionary.
4. The **usecmap** operator (CMap files that use another) appears after the **begincmap** operator and before any range operators.
5. The **begincodespacerange** operator must be the first range operator in the file. It must appear after the **begincmap** operator. It is implicit in a CMap file that uses another and in a rearranged font.
6. The **endcmap** operator must be the final operator in the file. It and the **begincmap** operator bracket the CMap dictionary.

7.4 Operator Details

This section contains detailed information about the operators supported in PostScript language CMap files. If the characters **(RF)** appear at the far right of the operator definition, it means that the operator applies *exclusively* to rearranged fonts.

beginbfchar *int* **beginbfchar** –
endbfchar *srcCode dstCode* **endbfchar**–
srcCode /dstCharname **endbfchar** –

The **beginbfchar** and **endbfchar** operators map *int* number of individual input codes (*srcCode*) to a corresponding number of individual character codes (*dstCode*) or character names (*dstCharname*), where *int* can be ≤ 100 . The *dstCode* can be drawn from font programs of Type 0, 1, or 3; *dstCharname* can be drawn from font programs of Type 0 or 1. The base font that contains the glyphs must have been specified by a previous **usefont** call.

srcCode and *dstCode* must be specified as hexadecimal strings. *dstCharname* must be a PostScript language name object.

There can be a maximum of 100 lines in each **~bfchar** set.

Use the **~bfchar** operators when the mappings to be described are organized *noncontiguously*, for example, when you want to define the relationship between *sets* of individual input codes and individual glyphs rather than contiguous *ranges* of codes and glyphs.

Example

```
2 usefont
4 beginbfchar
<27> /quotesingle
<5c> /yen
endbfchar
```

Errors

stackunderflow, syntaxerror, typecheck

See Also

beginbfrange, usefont

beginbfrange *int* **beginbfrange** –
endbfrange *srcCodeLo srcCodeHi dstCodeLo endbfrange* –
srcCodeLo srcCodeHi [/dstCharName₁../dstCharName_n] **endbfrange** –

The **beginbfrange** and **endbfrange** operators map *int* number of ranges of input codes to a corresponding range of character codes or names, where *int* can be ≤ 100 . The argument *srcCodeLo* is the start of a given range of input codes; *srcCodeHi* is the end of that range. The argument *dstCodeLo* is the start of the corresponding character code range; there is no need to specify the upper limit of the range. Alternatively, an array of character names can be specified to correspond to the range of input codes. All character names specified in this way must be fully enumerated.

Values for *srcCodeLo* and *srcCodeHi* must be in hexadecimal notation. The *dstCode* can be drawn from font programs of Type 0, 1, or 3; *dstCharname* can be drawn from font programs of Type 0 or 1. The base font that contains the glyphs must have been specified by a previous **usefont** call.

There can be a maximum of 100 lines in each **~bfrange** set.

Use the **~bfrange** operator when the mappings to be described are organized in contiguous ranges.

Errors

stackunderflow, syntaxerror, typecheck

See Also

beginbfchar, usefont

begincidchar *int* **begincidchar** –
endcidchar *srcCode dstCID* **endcidchar** –

The operators **begincidchar** and **endcidchar** map *int* number of individual valid input codes to a corresponding number of individual character IDs, where *int* can be ≤ 100 . The argument *srcCode* is an input code expressed as a hexadecimal string; the argument *dstCID* is a character ID expressed as an integer.

There can be a maximum of 100 entries in each **~cidchar** set.

Use the **~cidchar** operators when the mappings to be described are organized *noncontiguously*, for example, when you want to define the relationship between sets of *individual* input codes and *individual* character IDs rather than contiguous *ranges* of codes and character IDs.

Errors

stackunderflow, rangecheck, typecheck

See Also

begincidrange

begincidrange *int* **begincidrange** –
endcidrange *srcCodeLo srcCodeHi dstCIDLo* **endcidrange** –

The **begincidrange** and **endcidrange** operators map *int* number of ranges of input codes to a corresponding range of character IDs, where *int* can be ≤ 100 . The argument *srcCodeLo* is the start of a given range of input codes, and *srcCodeHi* is the end of that range. The argument *dstCIDLo* is the start of the corresponding range of character IDs; there is no need to specify the upper limit of the range. Ranges may overlap, but succeeding ranges supercede previous ranges. Ranges *should* appear in ascending order. Values for *srcCodeLo* and *srcCodeHi* must be in hexadecimal notation.

There can be a maximum of 100 entries in each **~cidrange** set.

Use the **~cidrange** operators when the mappings to be described are organized in contiguous ranges.

Example

```
100 begincidrange
    <20> <7e> 231
    <8140> <817e> 633
    <8180> <8188> 696
    <8189> <8189> 7478
    <818a> <81ac> 706

    << 90 ranges omitted >>

    <e080> <e092> 5563
    <e093> <e093> 7838
    <ea80> <ea9c> 7443
    <ea9d> <ea9d> 7886
    <ea9e> <ea9e> 7473
endcidrange
```

Errors

stackunderflow, rangecheck, typecheck

See Also

begincidchar

begincmap – **begincmap** –
endcmap – **endcmap** –

These operators must enclose that portion of the CMap file that contains the code mapping information. They produce objects in the CMap resource in VM that will subsequently be used to map character codes to font IDs and character selectors.

begincodespacerange *int* **begincodespacerange** –
endcodespacerange *srcCodeLo srcCodeHi* **endcodespacerange** –

These operators define as valid *int* number of ranges of input codes, where *int* can be ≤ 100 . The arguments *srcCodeLo* and *srcCodeHi* are expressed in hexadecimal notation.

Input codes can consist of one, two, three, or more hexadecimal bytes. The number of bytes in the input code establishes the dimensionality of the codespace range or *region*. For example, one-byte input codes describe a

linear region of valid input codes, two-byte codes describe a rectangular region of valid input codes, and so forth. Section 5 describes codespace more extensively.

There can be a maximum of 100 entries in each `~codespacerange` set.

Codespace regions need not be contiguous but cannot overlap. The definition of the codespace must precede any mapping of input codes to characters.

In rearranged fonts, the codespace of the template font defines the codespace for the rearranged font.

Example

```
4 begincodespacerange
  <00> <80>
  <8140> <9FFC>
  <A0> <DF>
  <E040> <FBFC>
endcodespacerange
```

Errors

stackunderflow, rangecheck, typecheck

beginnotdefchar *int* **beginnotdefchar** –
endnotdefchar *srcCode dstCID* **endnotdefchar** –

The operators **beginnotdefchar** and **endnotdefchar** map *int* number of individual valid input codes to a corresponding number of individual character IDs, where *int* can be ≤ 100 . Each character ID references a notdef character. A font developer can use the `~notdefchar` operators to map otherwise valid input codes to specific notdef characters within the CIDFont.

The argument *srcCode* is an input code expressed as a hexadecimal string; the argument *dstCID* is a character ID expressed as an integer.

Every CID-keyed font must have at least one notdef character defined. This notdef character is referred to by CID 0. Invalid input codes (input codes which are outside of codespace) are automatically mapped to CID 0.

There can be a maximum of 100 lines in each `~notdefchar` set.

The effect produced by showing a notdef character is left to the discretion of the font designer.

Note: The conditions under which a notdef character is shown are discussed in section 5.

Errors

stackunderflow, rangecheck, typecheck

See Also

beginnotdefrange

beginnotdefrange *int* **beginnotdefrange** –
endnotdefrange *srcCodeLo srcCodeHi dstCID* **endnotdefrange** –

The operators **beginnotdefrange** and **endnotdefrange** map *int* number of ranges of valid input codes to a corresponding number of character IDs, where *int* can be ≤ 100 . Each range of input codes maps to the same character ID; if a character ID falling within the notdef range is presented to the CMap resource instance, which for some reason (such as an empty interval) cannot show a glyph, then the notdef character defined here will be shown instead. A font developer can use the **~notdefrange** operators to map ranges of otherwise valid input codes to specific notdef characters within the CIDFont.

The argument *srcCodeLo* is the start of a range of input codes; *srcCodeHi* is the end of that range. Input codes are expressed as hexadecimal strings. *dstCID* is the character ID to which all input codes in the range are mapped. It is expressed as a number.

Every CID-keyed font must have at least one notdef character defined. This notdef character is referred to by CID 0. Invalid input codes (input codes that are outside of codespace) are automatically mapped to CID 0. An undefined CID may occur if Supplement numbers are not the same between a CMap and a CIDFont file; a notdef character results.

There can be a maximum of 100 entries in each **~notdefrange** set.

The effect produced by showing a notdef character is left to the discretion of the font designer.

Note: The conditions under which a notdef character is shown are discussed in section 5.

Example

```
2 beginnotdefrange
  <00> <1f> 1
  <fc> <ff> 231
endnotdefrange
```

Errors

stackunderflow, rangecheck, typecheck

See Also

beginnotdefchar

beginrearrangedfont */newFontName [component fonts array]* **beginrearrangedfont** – (RF)
endrearrangedfont **endrearrangedfont** –

The **beginrearrangedfont** and **endrearrangedfont** operators bracket the definition of a rearranged CID-keyed font, and can appear only in CMap files describing a rearranged font. Rearranged fonts are discussed in section 6.

The parameter *newFontName* is the name given to the resulting rearranged font. The *component fonts array* is a list of fonts that contribute characters to the rearranged font. The zero-th element of the component fonts array is the *template font*, which controls the codespace of the rearranged font. Succeeding elements of the array are font names, each of which contain characters that will be borrowed for specific rearrangements.

The operators **usefont** and **beginusematrix** operate on fonts selected from this array by position; each takes an integer argument that is an index into this array.

Each of the fonts named in the component fonts array must be present on the PostScript interpreter for a rearranged font to work.

If you wish to use the same component font in two (or more) different ways, it must appear in the **beginrearrangedfont** array more than once. For example, if you wish to use a component font both with and without a transformation matrix, that font must appear twice in the array.

To do a rearrangement, you must know the codespaces of the template font and all component fonts.

Example

```
%ADOSTartRearrangedFont
/Jun101-Light-K-G-R-83pv-RKSJ-H

[ /Jun101-Light-83pv-RKSJ-H
  /Poetica-ChanceryIV
  /ShinseiKai-CBSK1-83pv-RKSJ-H
  /FutoGoB101-Bold-83pv-RKSJ-H
  /FutoMinA101-Bold-83pv-RKSJ-H
  /HSMInW3Gai30
] beginrearrangedfont
```

Errors

stackunderflow, **syntaxerror**, **typecheck**, **VMerror**

See Also

usefont, **beginusematrix**

beginusematrix *fontID* **beginusematrix** –
endusematrix [*a b c d tx ty*] **endusematrix** –

These operators bracket the specification of a transformation matrix to be applied to the font within the component fonts array, specified by *fontID*. They provide a mechanism to apply rotational, obliquing, narrowing, and expanding transformations, and baseline translations to a given font. The effect of the **beginusematrix** and **endusematrix** operators is equivalent to applying **makefont** to the base font, using the same matrix.

The units of the transformation matrix are expressed in character coordinate space. For example, to move a baseline up by .150 em, you would use the matrix [1 0 0 1 0 0.15], with *ty* being 150/1000. See *Adobe Type 1 Font Format* for a complete explanation of the FontMatrix.

The **~usematrix** operators can only be used after the **beginrearrangedfont** operator has already been called, which specifies the component fonts array for a particular rearrangement. The *fontID* argument to **beginusematrix** is a zero-based index into this array; however, the value of *fontID* must be greater than 0 (the zero-th element is a template font). The **~usematrix** operators fail if there has been no prior call to **beginrearrangedfont** (**syntaxerror**) or if *fontID* is out of range (**rangecheck**).

Example

This example substitutes the “A” hiragana and katakana characters in Ryumin-Light-83pv-RKSJ-H with those of MyKana-83pv-RKSJ-H. The baseline of MyKana-83pv-RKSJ-H is raised slightly using the **~usematrix** operators.

```
/MyKanjiFont
[/Ryumin-Light-83pv-RKSJ-H /MyKana-83pv-RKSJ-H] beginrearranged-
font
1 beginusematrix [1 0 0 1 0 0.1] endusematrix
1 usefont
2 beginbfchar
<82a0> <82a0> % hiragana-"A"
<8341> <8341> % katakana-"A"
endbfchar
endrearrangedfont
```

Errors

stackunderflow, typecheck, syntaxerror, rangecheck

usecmap */CMapName* **usecmap**

The **usecmap** operator allows one resource instance to refer to the structures already created by another. The amount of VM saved can be substantial. The argument */CMapName* is the name of the CMap resource instance that is being referred to. The **usecmap** operator incorporates the codespace and code mappings from that file into its own.

The **usecmap** operator must precede any specification of code mappings.

If the CMap file *being used* contains character code mappings that have been described in the *using* file, the definitions in the *using* file are ignored (essentially, they are *overridden*). CMap files can be nested to five levels.

Errors

stackunderflow, typecheck, undefinedfilename, VMerror

usefont *fontID* **usefont** –

The **usefont** operator is used after the **beginrearrangedfont** operator to specify a font to be used for a series of subsequent operations. The **beginrearrangedfont** operator specifies an array of component fonts for a particular rearrangement. The *fontID* argument to **usefont** is a zero-based index into this array (the zero-th element is the template font).

In compatibility mode operation, the **usefont** operator fails if there has been no prior call to **beginrearrangedfont** (**syntaxerror**) or if *fontID* is out of range (**rangecheck**).

Errors

stackunderflow, typecheck, rangecheck

Appendix A: Installing CID-Keyed Fonts on PostScript Interpreters

A.1 Introduction

This appendix describes how to install CID-keyed font resources, related resources, as well as files that are related to row-organized fonts, on a PostScript interpreter.

Fonts of CIDFontType 0 are designed to be installed into a file system, such as that on an external storage device (hard disk) of a printer or host computer, or to be downloaded to VM. CID-keyed fonts are generally installed by an installation program. This appendix is intended for developers of such programs.

A.2 PostScript Interpreter Requirements

There are two requirements for an interpreter to be able to run CID-keyed fonts of CIDFontType 0. The first requirement is that the PostScript interpreter must support Type 0, or *composite* fonts. A PostScript interpreter can support Type 0 fonts if it is a Level 1 interpreter with the Composite Font extensions, or if it is a Level 2 interpreter.

Note While all Level 2 interpreters have the necessary language capability to interpret CID-keyed fonts, the character caching capability of version 2010 PostScript interpreters may be inadequate for large Asian fonts. It is also advised that software doing downloading or installation should check to make sure the printer has at least 1 megabyte of RAM.

The second requirement is that the PostScript interpreter have access to a set of CID-keyed fonts and a set of files referred to as the *CID Support Library* (CSL). Some of these other files are executed during (Sys/Start) or (Sys/Bootlist) execution. Currently, there are about 40 files in the CSL, and the contents of the files are explained in the *contents.txt* file supplied with the CSL.

The CID Support Library contents are of two types:

- CSL core modules

- Language specific support. This part includes items such as CMap files for each language, for example for the Adobe-Japan1-2, or Adobe-GB1-0 (Simplified Chinese) character collections.

The CID Support Library software is available to developers for bundling with font installation programs, subject to certain licensing conditions. For more information, contact the Adobe Developers Association.

A.3 Categories of Installation Files

The files which comprise an installable CID-keyed font and CID Support Library which a font installation program might install, are known as *candidate files*. That is, they are candidates to be installed.

Files which already exist in the PostScript interpreter's file system, and which have the same name as candidate files, are called *target files*.

Target files may be partitioned into two categories:

- A. Files to be modified, or added if not present.
- B. Files to be replaced.

The first category consists of the target files:

(Sys/Start)
(Sys/Bootlist)

These are referred to as *Category A files* throughout this appendix. These target files may require modification. If none of the target files are present, then a new candidate file may be installed.

The second category consists of all remaining target files. These files are replaced or added as needed; they are never partially changed. These files are referred to as *Category B files* throughout this appendix.

The complete list of files in Category B may vary according to the particular release of the CID Support Library. Please see the Release Notes for a given release for the specific enumeration of these files.

A.4 Installation Environment

File Names

PostScript string syntax is used in this document to refer to both candidate and target file names. See the *PostScript Language Reference Manual, Second Edition* to determine how PostScript strings are interpreted as file names.

File Searching and Devices

When the installation algorithm described in this appendix refers to searching for a file, it refers to locating a file when there may be many attached devices. On PostScript interpreters, devices are prioritized for file searching. External devices, such as printer hard disks are searched first; then cartridge devices are searched second; finally, ROM devices are searched last. The specific search order is determined by numbering each device (explained below), where lower numbers indicate higher search priority. The first device checked that has the searched file ends the search. The number of a device is determined as follows:

For Level 2 interpreters, the operator **currentdevparams** returns a dictionary having a key `/SearchOrder` indicating the number for this device.

For Level 1 interpreters, the number is typically that of the SCSI identification number. This is the same number that is reported during the **devforall** operator as in `(%disk<n>%)`, where the `<n>` is the SCSI ID.

CID Support Library Installation and Low-order Device

Installation of the CID Support Library can be performed to a specified device or to the *low-order device*, which is the device with the highest search order priority. This choice is to be made before employing the installation algorithm described in the sections below. In the instructions that follow, the specified device for installation is referred to as the *selected device*. In *all* cases, the selected device *must* be a writable device.

If the selected device is not the low-order device, then the installation algorithm must know whether or not all identically named files on lower-order writable devices relative to that of the target file shall be deleted. This condition is called the *Deletion Toggle* and is true if deletion shall occur; otherwise, it is false. This is necessary if the CID Support Library installation is expected to run in the hardware configuration as of installation time. Were this not to happen, the installed files on the selected device would never be referenced because the identically named files on lower-order devices would be found instead.

Naturally, files on read-only devices cannot be deleted; but, these files are typically cartridge or ROM devices and are, therefore, found later than those of the selected device. If there are read-only devices that have a higher search priority than the selected device, these shall be changed to have a lower search priority before employing the installation algorithm. If doing that is not possible, then proper execution of the CID Support Library in the hardware configuration, as of installation time, might not be possible. This circumstance is not expected to occur, but has been addressed here for completeness.

Sometimes the purpose of an installation is to put a complete CID Support Library onto a specific device. This condition is called the *Search Target Device Only Toggle*. If it is set to true, then the installation program shall search for target files only on the current device. If false, it shall search for target files on all devices, in search priority order.

Version Checking

Version checking is the process of determining whether or not a candidate file will be installed. It consists of checking within the target file on the selected device for DSC comments indicating the version number associated with that file. If the target file is not on the selected device, but is on some other device, then the file on the low-order device is used for version checking.

```
%%BeginResource: <category> (<assignedName>
...
%%Version: <versionNumber> [<revisionNumber>]
...
%%EndResource
```

Each file of the CID Support Library shall contain lines as shown above. For files that define PostScript language resources, the <category> portion above is the resource category; all other files use “file” as the <category>.

For files of Category A, the <assignedName> is (CID Support Library) for (Sys/Start) and (CID Support Library Bootlist) for (Sys/Bootlist). For files of Category B, the <assignedName> is the resource instance or name of the file. Note that each occurrence of <assignedName> is expressed as a PostScript string object.

The %%Version comment identifies the version number for the block. This comment line can appear anywhere within a block of code delimited by

```
%%BeginResource and %%EndResource
```

The value of <versionNumber> is a real number, and the optional <revisionNumber> is an integer. A typical example might be:

```
%% Version 1.1 2
```

Where 1.1 is the version number, and 2 is the revision number. If the revision number is omitted, it is assumed to be zero. The revision number should be taken into consideration if the version number of two files are identical. For example:

```
%% Version: 1.402 2
```

or

%% Version 1.402

are older than:

%% Version 1.402 3

The following examples show how a block of DSC comments describes a resource with a version number.

```
%%BeginResource: file (CID Support Library)
...
%%Version: 1.01 2
...
%%EndResource

%%BeginResource: file (FS/Level2CID)

...
%%Version: 1.01 2
...
%%EndResource

%%BeginResource: CMap (83pv-RKSJ-H)
...
%%Version: 1.01 2
...
%%EndResource
```

Installation Software and File List

The list of files in Category B may vary for different releases of the CID Support Library. Installation software shall not depend on a specific list of files. The list of Category A files is not expected to change.

A.5 Prior to Installation

Before performing the installation of the CID Support Library, the installation algorithm must determine the following information:

- Deletion Toggle
- Search Target Device Only Toggle
- Selected Device

It is also important for the installation software to determine if there is sufficient space on the selected device for the installation of the CID Support Library.

A.6 Installation of Category A Files

Determining the Target File

For Category A, the algorithm described below is to be performed. Note that this algorithm selects one file to operate on, and that this operation is not applied to every file in the category.

Determine the target file to be modified (select the first that applies):

1. If there is a (Sys/Bootlist) file, then modify it using the resource identified by **%%BeginResource: file (CID Support Library Bootlist)**; else,
2. If there is a (Sys/Start) file, then modify it using the resource identified by **%%BeginResource: file (CID Support Library)**.
3. If neither of the above files are found, then copy the entire candidate file named (Sys/Start) to the selected device. Take no further action for this category.

Note If both files exist, only the one file selected as shown above is to be acted upon; the other file will not be used or acted upon in any way.

Checking Version Information

Check the version information and determine if the file requires modification. If it does, continue; otherwise, no further action is necessary for Category A files.

Modifying the File

If the file contains the DSC comments:

```
%%BeginResource: file (CID Support Library)
...
%%EndResource
```

or, in case of (Sys/Bootlist):

```
%%BeginResource: file (CID Support Library Bootlist)
...
%%EndResource
```

then, replace that portion of code with the code that is bracketed in the same way in the candidate file. The replacement code must include the DSC comments shown above.

If the file does not contain the DSC comments above, then the replacement text is to be inserted according to the following heuristic:

- A. The DSC comments that follow may bracket the portion of the file that may be modified by installation software:

```
%ADOBeginCustomStartup
...
%ADOEndCustomStartup
```

If only one of the comments is present in a file, then:

1. if the omitted comment is %ADOBeginCustomStartup, then the file may be modified from the beginning of the file to immediately before the %ADOEndCustomStartup comment; and,
 2. if the omitted comment is %ADOEndCustomStartup, then the file may be modified from immediately following the %ADOBeginCustomStartup comment to the end of file.
- B. If the DSC comment %ADOEndCustomStartup is in the file, then the insertion is to appear immediately before this comment. If there is no such comment, then the insertion is to occur as follows:
1. if the file is (Sys/Bootlist), then the insertion is to occur immediately before the last line of the file.
 2. if the file is (Sys/Start), then the insertion is to occur immediately before the PostScript string token "(Usr/Start)", if present. However, this is true only if that token is within the customizable portion of the file. If there is no such string token, or if that token is not within the customizable portion of the file, then the insertion is to take place at the end of the customizable portion of the file. That is, the insertion is to occur immediately before the comment %ADOEndCustomStartup, if present; otherwise, the insertion is to occur after the last line of the file.

Note Presently, no product in the field has the DSC comments: %ADOBeginCustomStartup and %ADOEndCustomStartup. The algorithm as described by A and B above is a heuristic to be used when these comments are not found. To ensure that the algorithm is deterministic, the files in Category A can be preconditioned so that these comments are inserted. If an insertion location other than that shown above is desired, the comments:

```
%%BeginResource: file (CID Support Library)
%%Version: 0.0
%%EndResource
```

or the comments:

```
%%BeginResource: file (CID Support Library Bootlist)
%%Version: 0.0
%%EndResource
```

can be inserted in an existing (Sys/Start) file. Doing this will ensure that replacement will occur in the location where these comments are found.

Example 25 Summary of Sys/Start modifications

```
<<Sys/Start header comments here >>

%%BeginResource: file AdobeCompositeFontSupport
/languagelevel where { pop languagelevel 2 ge } { false } ifelse
{ { (FS/Level2) } { (FS/Level1) } ifelse run } stopped clear
%%EndResource
%%BeginResource: file AdobeCIDKeyedFontSupport
{ /CIDInit /ProcSet findresource } stopped clear
{ /83pv-RKSJ-H /CMap findresource } stopped clear
%%EndResource

<<Remainder of unmodified Sys/Start file (if any) follows here >>
```

A.7 Installation of Category B Files

For Category B, the following algorithm is to be followed for each file in this category:

1. Determine the file to be modified. (See *File Searching and Devices* above.)
2. Check the version information and determine if the file requires replacement. If it does, continue; otherwise, no action for this file. Continue checking other files for possible replacement.
3. If the file requires replacement, then it is to be removed and replaced with the replacement file.

Appendix B: ATM-J Compatibility for CID-Keyed Fonts

This appendix explains how to install CID-keyed fonts on a Macintosh computer for use with Adobe Type Manager software, Japanese Edition (ATM-J). This appendix is specific to both Macintosh and ATM-J version 3.5 or greater.

B.1 Installing CID-Keyed Fonts on the Macintosh

When CIDFont and CMap files have been properly installed, ATM-J can parse and make use of them directly, without recourse to other system support files on the system such as those detailed in Appendix A.

As explained in section 2, the CIDFont file contains glyph data indexed by character ID. The CMap file specifies the subset of that character collection to be used, called the *character set* (or *charset*). A CMap file also imposes an encoding on that subset, in which character codes are mapped to CIDs.

B.1.1 CIDFont Files

As shown in section 3, a CIDfont file is essentially an ASCII text file with character description and other data at the tail. When transferred to or created on the Macintosh, it occupies the *data fork* of a Macintosh file. For compatibility with ATM-J, a CIDFont *must* occupy the data fork; ATM-J never references the resource fork.

Note So that the correct file icon can be displayed on the desktop, the resource fork of a CIDFont file should include BNDL, FREF, and ICN# resources. Each CIDFont file installed on a Macintosh must also have its file type set to LWFN. ATM-J never references the file creator's signature.

For System 7.1 or later, install CIDFonts in the Fonts folder; for System 6.x, install CIDFonts in the System folder. When searching for CIDFonts, ATM-J looks first in the System folder, and then checks the Fonts and Extensions folders, if present.

For ATM-J to recognize CID-keyed fonts, you must also install the corresponding screen font resources. Screen font resources are described in technical documentation available from Apple Computer, Inc.

B.1.2 CMap Files

As with CIDfont files, a CMap file is an ASCII text file. When transferred to or created on the Macintosh, it also must occupy the data fork of a Macintosh file. ATM-J never references the resource fork, and it does not check the file type or creator's signature of CMap files. Adding specific resources to the resource fork is not necessary.

In the past, Adobe composite fonts installed on the Macintosh resulted in the creation of a *Common* folder within the *System* folder. This contains system support files for composite fonts. Other folders within the *Common* folder include

:encodings
:charstrings
:Generic

Adobe recommends that during the installation process, CMap files be copied to a folder called *CMaps*, within the *Common* folder. ATM-J looks for CMaps only in the *:Common:CMap* folder; it expects the *Common* folder to be located in the *System* folder.

Note So that the correct file icon can be displayed on the desktop, the resource fork of a CMap file should include *BNDL*, *FREF*, and *ICN#* resources. Each CMap file installed on a Macintosh must also have its file type set to *LWFN*. ATM-J never references the file creator's signature.

B.2 Naming Conventions

File naming is important to the Macintosh and to ATM-J. Font names must be unique, and the Macintosh derives font names from filenames in a particular fashion. For information on font naming conventions, see Adobe Technical Note #5088, *Font Naming Issues*.

B.3 Parsing Considerations

ATM-J does not include a complete PostScript interpreter, and consequently parses CIDFont and CMap files in a simple fashion. To remain compatible with ATM-J, such files must strictly conform to the document structuring conventions, the syntax and lexical conventions as explained in Sections 3 through 7, and the additional ATM-J parsing rules outlined here. All CIDFont and CMap examples in this document do conform and exhibit the properties necessary for them to be parsable by ATM-J.

ATM-J (and other simplified PostScript language parsers) generally separate the text of a CID-keyed font program into *tokens* according to PostScript language rules as defined in *PostScript Language Reference Manual, Second*

Edition. Comments are ignored when looking for tokens. Parsers such as ATM-J check tokens for certain keywords when they occur at the “top level” of code (not when they are contained in procedure bodies), and take action based on those keywords. For these reasons, for CIDFont and CMap files to be compatible with ATM-J, they must conform to these rules:

- Individual tokens and charstrings may not exceed 65535 characters in length.
- Most keywords are names that are associated with values in a dictionary; the initial portion of a CIDFont program is assumed to contain names to be inserted into a CIDFont dictionary.
- If the keyword **eexec** appears, the text following it *must* be encrypted. No assignments of values to names may occur in the plain text that follows the encrypted portion. See *Adobe Type 1 Font Format* for more information about **eexec** encryption.
- When a simple value (integer, real, string, or Boolean) is associated with a name in a dictionary, that value must *follow* the name immediately as the next token.

For example, Boolean values may be only the tokens *true* or *false*. Simple values, such as integers, must explicitly be written after a name—they may not be computed by a sequence of PostScript language constants and operators.

```
Right way:           /CIDFontType 1 def
Wrong way:           1 /CIDFontType exch def
Wrong way:           /CIDFontType 2 1 sub def
```

Even though both “wrong” ways are legal and equivalent PostScript language code, they do not conform to the parsing rules required by ATM-J.

- When an array is expected as a value, the array must immediately follow the name to which it is assigned. An array must begin with either [or { and terminate with the corresponding } or]. Numeric contents must occur as single tokens within the array delimiters.
- When a **begin** operator occurs to change the current dictionary, it must end with one and only one occurrence of the corresponding **end** operator. **Begins** and **ends** must be accurately paired.

B.4 Miscellaneous Notes for Macintosh ATM before version 3.5

- ATM (before version 3.5) does not support CID fonts.

B.5 Miscellaneous Notes for Macintosh ATM version 3.5

- ATM-J supports Shift-JIS-encoded CMap files. It does not currently support JIS or EUC-encoded CMaps.
- All double-byte characters must be of fixed width at 1000/em (full width).
- ATM-J does not support the **usecmap** operator. Because of this, ATM-J does not support the vertical variant CMaps (~V).
- ATM-J does not support CMap operators that specify characters by name.
- ATM-J currently does not support any CMap range operations in which more than the last byte varies between two input codes. For ATM-J version 3.5 to work properly, only the last byte in a range operation can vary.
- When parsing rearranged fonts, ATM-J is particularly sensitive to the following document structuring comments:

```
%ADOResourceSubCategory: RearrangedFont  
%ADOStartRearrangedFont
```

These comments must be used as documented in Section 6.

Appendix C: Obtaining CID Information

C.1 Support for CID-keyed Font Development

Support for developing CID-keyed fonts is available from the Developer Relations department of Adobe Systems. Please use the addresses on the cover of this document.

All contact for development information should be with the Developer Relations department, but there are several sources of information within this department. If you require a UniqueID number or information about UniqueIDs, it is important to address correspondence or fax transmission to *Attention: Unique ID Coordinator* within the Developer Relations group.

Table C.1 shows whom to contact at Adobe Systems for your CID-keyed font development needs.

Table C.1 *Whom to contact at Adobe Systems*

<i>Requirement</i>	<i>Contact</i>
UIDBase number	Unique ID Coordinator
XUID organization number	Unique ID Coordinator
Registry Strings	Unique ID Coordinator
CIDInit Procset	Developer Relations
System Support files	Developer Relations
CJK Language CMap Files	Developer Relations
All related technical notes and character collection documents	Developer Relations

Appendix D: Font Naming and Unique ID Numbers

D.1 CID Font Naming

It is important that CID-keyed fonts be named in a way that helps the font machinery find and execute them easily. This section explains the recommended way to name CID-keyed fonts.

The **findfont** operator looks for the name of a CID-keyed font in two pieces. One piece is *CIDFontName*, which is looked for in the *CIDFont/* directory. The other piece is the *CMapName*, which is looked for in the *CMap/* directory. The two parts fit together like this:

`<CIDFontName>--<CMapName>`

where `<CIDFontName>` is the name of the CIDFont file and `<CMapName>` is the name of the CMap to be used with that CIDFont. The two parts are separated by a delimiter, which should be a double hyphen. (For backwards compatibility, a single hyphen is allowed, but Adobe encourages the use of the double hyphen.) For example, the font *Ryumin-Light-90pv-RKSJ-H* is made up of the two files *Ryumin-Light* and *90pv-RKSJ-H*. Another font *Mincho-Light--83pv-RKSJ-H* (note the use of the preferred double hyphen) is made up of *Mincho-Light* and *90pv-RKSJ-H*.

The two filename parts are themselves made up of smaller elements, which are outlined in Adobe Technical Note #5088, *Font Naming Issues*. The main purposes of such a font naming strategy however, can easily be summed up:

- It ensures that complex CID-keyed font names work properly with the redefined **findfont** operator.
- It provides guidelines for names that impart information about the font and its character set and encoding.
- It ensures that each font name is unique, which is necessary for correct handling.

In general, CIDFont names describe the glyphs that make up a particular collection; CMap names describe a particular combination of character set and encoding that is *font-independent*.

D.2 Calculating Unique IDs

Individual CMap files consume identification numbers based on the nature of the ranges specified in the *codespace* definition. Codespace represents the set of valid input codes. See section 5 for an explanation of how codespace is defined. Unique ID numbers have been pre-calculated for all standard Japanese CMap files; a developer needs to figure the count only when creating a new CMap. Further, calculating Unique IDs using UIDBase and UIDOffset, as shown here, is necessary only for compatibility mode operation; in native mode operation, the CMap and CIDfont are identified by XUID, which requires no calculation.

For this example, the ranges specified for codespace are assumed to be <00> to <80>, <8140> to <9FFC>, <A0> to <DF>, and <E040> to <FCFC>.

Unique ID numbers are assigned on a per-row basis; the total count of ID numbers consumed is equal to the count of one-byte ranges plus the count of two-byte codes (within a given listed range) that differ only in the last byte.

Note There can be three-byte, four-byte, and greater ranges. However, the rule of thumb is the same: the count of codes within a given listed range that differ only in the last byte.

In this example, there are 62 unique ID numbers consumed. There are two one-byte ranges (<00> to <80> and <A0> to <DF>). Between <8140> and <9FFC> there are 31 ranges that differ only in the last byte. <8140> to <81FC> is the first such range, <8240> to <82FC> is the second, and so forth. The range <E040> to <FCFC> includes 29 such ranges. The count of the numbers consumed is therefore $2 + 31 + 29 = 62$.

D.2.1 Assigning the ID Count

Because additional characters may be added to a collection after its initial production, Adobe encourages developers to “pad” the total count to allow for future expansion. For example, while the CMap file *Ext-RKSJ-H* consumes 62 identification numbers for caching, the number is “padded” to 70 to accommodate future additions.

Note Padding is also a good idea because once a developer has established a UID-Offset for a font marketed in the field, that number cannot be changed.

When assigning UIDOffset values to a group of CMap files that refer to a single CIDFont, enough room must be left between CMap files so that no identification numbers overlap. If overlapping occurs, bitmaps cached by a previous job may be obtained that reference the wrong glyph. This is a particular problem for service bureaus where cached characters might be written to disk and remain there during subsequent jobs.

Table D.1 shows the standard CMap files for the CID-keyed Japanese character collection provided by Adobe, their ID counts, and their UIDOffset values.

Table D.1 *UIDOffset values*

<i>Character Collection</i>	<i>IDs CMap for</i>	<i>Count Required</i>	<i>UID Used</i>	<i>Offset</i>
Adobe-Japan1-0	83pv-RKSJ-H	63	70	0
	Ext-RKSJ-H	62	70	70
	Add-RKSJ-H	62	70	140
	RKSJ-H	62	70	210
	H	94	100	280
	EXT-H	94	100	380
	NWP-H	94	100	480
	Add-H	94	100	580
	EUC-H	96	100	680
	Add-RKSJ-V	5	10	780
	Add-V	5	10	790
	EUC-V	3	10	800
	Ext-RKSJ-V	5	10	810
	Ext-V	5	10	820
	NWP-V	6	10	830
	RKSJ-V	4	10	840
	V	3	10	850
	Hankaku	1	2	860
	Hiragana	1	2	862
	Katakana	1	2	864
Roman	1	2	866	
WP-Symbol	1	2	868	
Adobe-Japan1-1	90pv-RKSJ-H	63	70	870
	90pv-RKSJ-V	4	10	940
Adobe-Japan1-2	90ms-RKSJ-H	62	70	950
	90ms-RKSJ-V	7	10	1020

D.3 Miscellaneous Notes

- In compatibility mode, CMaps can only refer to a single CID or base font.
- In compatibility mode, Codespaces cannot exceed two bytes.

Appendix E: Changes since Earlier Versions

E.1 Changes in the 16 October 1995 version:

- Section 5.5, CMap File Naming Conventions, was added to explain how to ensure unique file names for custom CMap files.
- Appendix A was rewritten to reflect current installation techniques.
- Appendix D was added to contain the information on naming CID-keyed fonts and using UniqueID numbers that was previously in Appendix A.
- The UIDOffset Values table, now in Appendix D, has additional values for the Adobe-Japan1-1 and Adobe-Japan1-2 character collections

Index

Symbols

%! , comment conventions 13, 35, 52
%%BeginResource, comment conventions 13, 36, 53
%%DocumentNeededResources, comment conventions 36, 47
%%EndData, comment conventions 25
%%EndResource, comment conventions 26, 36, 53, 59
%%EOF, comment conventions 46
%%Include, comment conventions 13, 53
%%IncludedResources, comment conventions 47
%%IncludeResource, comment conventions 36
%%Title, comment conventions 13, 36
%%Version, comment conventions 14, 36, 53
%ADOSStartRearrangedFont, comment conventions 54
.notdef 44

A

Adobe Type Manager, Japanese Edition, see ATM-J
ATM-J 2, 25, 49, 60
compatibility with CIDFonts 81
parsing rules 82

B

Base font
Type 1 and Type 3 in rearranged font 56

beginbfchar 56, 63
beginbfrange 64
begincidchar 48, 65
begincidrange 48, 65
begincmap 37, 45, 47, 63, 66
begincodespacerange 63, 66
beginnotdefchar 67
beginnotdefrange 68
beginrearrangedfont 49, 53, 69
beginusematrix 54, 55, 70

C

candidate files 74
CDevproc 28
Character
code 4
codes 1, 4
collection 4, 5
definition 3
identifier (CID) 1, 4
name 4
names 1
selector 42
set 4, 32
Charset 4, 32
Charstring 19
data 9
definition of length 20
CID 0, default notdef character 5
CID procset, initializing 53
CID Support Library 73
CID Support Library installation 75
CIDCount 21, 28
CIDFont
and VM 9
CDevproc key 28
CIDCount key 28
CIDFontName key 28

- CIDFontType key 28
 - CIDFontVersion key 28
 - CIDMapOffset key 28
 - CIDSystemInfo key 28
 - comment conventions 13
 - compatibility with ATM-J 81
 - conceptual overview 3
 - data section 19, 27
 - encoding 33
 - example 10–13
 - FDArray key 29
 - FDBytes key 29
 - FontBBox key 30
 - FontInfo key 30
 - GDBytes key 30
 - handling subroutines 24
 - installing 73
 - internal organization 9
 - keywords 26
 - like and unlike PostScript language 8
 - Ordering key of CIDSystemInfo 29
 - Registry key of CIDSystemInfo 29
 - resource 4
 - resource instance 9
 - structural exceptions from Type 1 and Type 3 23
 - subset fonts 21
 - Supplement key of CIDSystemInfo 29
 - tutorial 8
 - UIDBase key 30
 - XUID key 30
 - CIDFont file
 - contents 6
 - Macintosh implementation 81
 - Macintosh resource fork 81
 - CIDFontName 15, 26, 28
 - CIDFontType 8, 15, 23, 28, 73
 - CIDFontVersion 14, 15, 28
 - CIDInit 26, 36, 53
 - procset 14
 - system support file 35
 - CIDInit 26
 - CID-keyed font files, see CIDFont and CMap
 - CIDMap 19, 27
 - and character ID 9
 - empty interval 22
 - first interval 21
 - last interval 21
 - CIDMapOffset 19, 28
 - cidrange 42
 - CIDSystemInfo 28, 37
 - CMap
 - character code map 4
 - conceptual overview 3
 - defined 32
 - installing 73
 - operators by group 62–63
 - resource 4, 45
 - resource dictionary 37
 - resource instance 33
 - CMap file 33, 45
 - comment conventions 35
 - errors and PostScript interpreter 60
 - Macintosh installation 82
 - Macintosh resource fork 82
 - naming convention 48
 - nomenclature 60
 - operator order 63
 - operators 59
 - purpose and contents 6
 - stand-alone 33–35
 - using another CMap file 46–47
 - CMapName 38, 45, 47
 - CMapType 38
 - CMapVersion 38, 47
 - Code mapping 42
 - range limitations 43
 - rearranged font 56
 - requirements 43
 - Codespace 33, 40, 88
 - limitation on ranges 40
 - requirements 40
 - Comment conventions
 - %! 35, 52
 - %%BeginResource 13, 36, 53
 - %%DocumentNeededResources 36, 47
 - %%EndData 25
 - %%EndResource 26, 36, 53, 59
 - %%EOF 46
 - %%Include 13, 53
 - %%IncludedResources 47
 - %%IncludeResource 36
 - %%Title 13, 36
 - %%Version 14, 36, 53
 - %%ADOSStartRearrangedFont 54
 - CIDFont file 13
 - CMap file 35
 - Compatibility mode 4, 26, 33, 90
 - Component font index 42
 - Component fonts 53
 - array 54
 - Component fonts, of rearranged font 49
 - Composite fonts 49, 73
 - Copyrights for CID-Keyed font programs 2
 - CPSI (Configurable PostScript Interpreter) 1
 - CSL (CID Support Library) 73
- ## D
- Data section of CIDFont, contents 27
 - defineresource 37
 - Deletion Toggle 75, 77
 - dictfull error 15, 37
 - Document structuring conventions 13
 - DPS (Display PostScript) 1
- ## E
- eexec 83
 - Empty interval 22, 44
 - Encoding of CIDFonts 33
 - endbfchar 56, 63
 - endbfrange 64
 - endcidchar 48, 65
 - endcidrange 43, 48, 65
 - endcmap 45, 63, 66
 - endcodespacerrange 66
 - endnotdefchar 67
 - endnotdefrange 68
 - endrearrangedfont 49, 53, 59, 69
 - endusematrix 54, 55, 70
 - Errors
 - CMap file operators 60
 - dictfull 15, 37
- ## F
- FDArray 9, 17, 22, 25, 27, 29
 - structure of 22
 - FDBytes 20, 22, 29
 - file names 74
 - file searching 75

findresource 14
findresource 14, 36, 53, 63
First interval of CIDMap 21
Font dictionary (FD) index 20
FontBBox 17, 30
FontInfo 30
FontInfo dictionary 19
FontMatrix 17

G

Gaiji characters
 adding to rearranged font 58
GDBytes 20, 30
Glyph
 data 9
 definition 3
 descriptor (GD) value 20
 rasterizing requirements 9

I

Initializing the CID procset 36, 53
installation of category A files 78
installation of category B files 80
Installing
 CIDFonts and CMaps 81
ioerror 32

J

Japanese Type 1 fonts, see Composite fonts

K

Keywords in CIDFont, required and optional 26

L

Last interval of CIDMap 21
Length of a charstring, definition 20

M

Macintosh
 naming conventions for CIDFont and CMap files 82

N

Naming conventions

 CIDFonts and CMaps on
 Macintosh 82
Native mode 33
 support for CID-keyed fonts 17
Native support (native mode) 4
Notdef
 characters 43
 CID 0 as default 5
 ranges 43
 when characters shown 44
Notdef characters 22

O

Ordering 5, 13, 16, 29, 36
Organization of a CIDFont 9
OtherSubr 24

P

Private dictionary 22, 24
Procset, initializing 36

R

Rearranged font 61
 adding gaiji characters 58
 contents 49
 defined 49
 example 50–52
 restrictions 49
Registry 5, 13, 16, 29, 36, 48
Resource instance
 CIDFont 9
Roman characters, replacing in
 rearranged font 55

S

SDBytes 24
Selected Device 77
selected device 75
Shift-JIS-encoded CMap files, and
 ATM-J 84
show 4
StartData 14, 25, 26, 31
 binary and hex arguments 26
 examples 32
 syntax 31
SubrMap 19, 24, 25, 27
SubrMapOffset 24, 25
Subroutine Descriptor (SD) values

24

Subroutine information in CIDFonts
 24
Subrs 24
Supplement 5, 13, 16, 29, 36
 no match between CIDfont and
 CMap 16
System support files 14, 35

T

target files 74
Template font 49, 61
 defined 49
Type 0 composite fonts 73
Type 1 fonts
 part of a CIDFont 42
 similarities to CIDFonts 8
 structural exceptions in CIDFonts
 23
Type 3 fonts
 part of a CIDFont 42
 similarities to CIDFonts 8
 structural exceptions in CIDFonts
 23

U

UIDBase 17, 18, 30, 39, 88
UIDOffset 17, 30, 39, 47, 88, 89
 count 39
Unique ID 30, 39
 calculating 88
 count 88
 numbers 17
 obtaining 85
usecmap 47, 63, 71
 ATM-J 84
usefont 54, 71
UserGaiji 59

V

Version Checking 76
Version control 5, 37
 CIDFont and CMap files 15

W

WMode 40, 47
Writing Mode, WMode 40

X

XUID 17, 18, 30, 39, 47, 88
 described 39
 special value 1000000 18