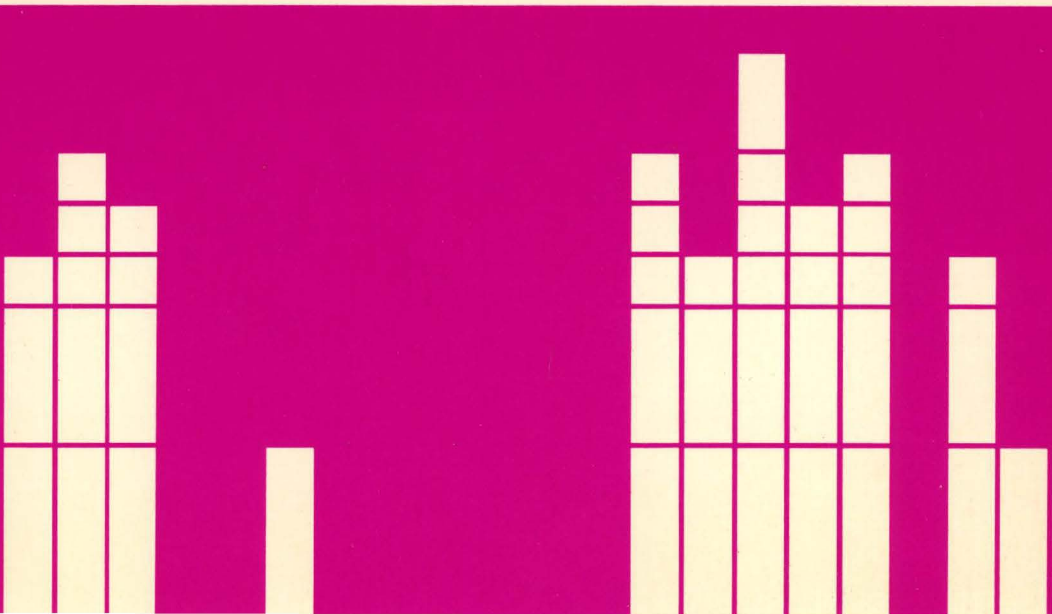
Publication Number
GC31-2070-0

4700 Finance Communication System

Controller Programming Library

Volume 5 Cryptographic Programming



4700 Finance Communication System

Controller Programming Library

Volume 5 Cryptographic Programming

Publication Number
GC31-2070-0

File Number
S370/4300/8100/S34-30

First Edition (July 1983)

This edition applies to Release 2 of the 4700 Finance Communication System and to all subsequent releases and modifications until otherwise indicated in new editions or Technical Newsletters (TNLs). This publication obsoletes the *IBM 4700 Finance Communication System Cryptographic Facilities Guide and Reference,* GC31–2045.

Changes occur often to the information herein; before using this publication to install or operate IBM equipment, consult the latest *IBM System/370 Bibliography of Industry Systems and Application Programs,* GC20–0370, for the editions that are applicable and current.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

Publications are not stocked at the address given below; requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. Address comments about this manual to IBM Corporation, Information Development, Department 78C, 1001 W.T. Harris Blvd., Charlotte, NC 28257 USA. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Preface

This is Volume 5 of the *4700 Controller Programming Library* — one of a set of six volumes for the 4700 programmer. Figure 0-1 on page iv summarizes the topics covered in the other volumes. All six volumes are available from your IBM representative or local branch office under a single order number (GBOF–1387).

Volume 5, Cryptographic Programming, describes the IBM 4700 Finance Communication System cryptographic facilities, and tells you how to use them.

You need this information if your responsibilities include the following:

- Designing, coding, or testing controller application programs that use the 4700 cryptographic facilities

- Establishing procedures for the generation, distribution, and control of cryptographic keys

- Installing a 4700 system that interacts with a cryptographic subsystem at the central site.

The first section of this manual introduces you to the 4700 cryptographic facilities. Subsequent sections tell you how to use the facilities to:

- encipher and decipher text
- generate and manage cryptographic keys
- authenticate messages
- process personal identification numbers (PINs).

This manual can be used both as a guide and as a reference. If you are already familiar with the concepts explained in this manual and simply want to refer quickly to a specific instruction, use the alphabetized descriptions in Chapter 10, "4700 Cryptographic Instructions." (There is also a complete alphabetic index at the back of this book.)

Before you can use the 4700 cryptographic facilities, you must know how to use the other 4700 facilities described in *Volume 1: General Controller Programming,* GC31–2066. Other related publications that you may need to consult are listed in the Bibliography at the end of this manual.

VOLUME 1: GENERAL CONTROLLER PROGRAMMING (GC31-2066)
- Programming Concepts
- Using the General Programming Instructions
- Coding Rules
- General Programming Instructions (Reference)
- General Machine Instruction Formats
- Parameter List Reference
- Status Codes, Program Check Codes, and Error Messages
- Programming Techniques for 3600 Compatibility

VOLUME 2: DISK AND DISKETTE PROGRAMMING (GC31-2067)
- Basic Diskette Programming
- Extended Diskette Programming
- Extended Disk Programming
- Disk and Diskette Programming Instructions (Reference)
- Disk and Diskette Status Codes
- Disk and Diskette Parameter List Reference
- Disk and Diskette Machine Instruction Formats

VOLUME 3: COMMUNICATION PROGRAMMING (GC31-2068)
- SNA/SDLC Host Communication Programming
- SNA/SDLC Communication Macros
- SNA/SDLC Communication Instructions (Reference)
- BSC3 Host Communication Programming
- BSC3 Communication Instructions (Reference)
- Communication Status Codes
- Communication Parameter List Reference
- Communication Machine Instruction Formats

VOLUME 4: LOOP AND DCA DEVICE PROGRAMMING (GC31-2069)
- General Protocols for Displays
- 4704 and 3604 Displays
- 3270-Compatible Displays
- 3606 and 3608 Financial Services Terminals
- General Protocols for Printers
- 4710 and 4720 Printers
- 3610, 3611, and 3612 Printers
- 3615 and 3616 Printers
- 3270-Compatible Printers
- 3624 Consumer Transaction Facilities
- Data Stream Mapping (DATSM) Protocols
- Device Status Codes
- Device Parameter List Reference

VOLUME 5: CRYPTOGRAPHIC PROGRAMMING (GC31-2070)
- Cryptographic Concepts and Facilities
- Enciphering and Deciphering Operations
- Generating and Exchanging Cryptographic Keys
- Authenticating Messages
- Validating and Translating PINs
- Using the Encrypting PIN Keypad
- Host Support Encryption Routines (BDKDPRS and BDKDES)
- Cryptographic Programming Instructions (Reference)
- System Cryptography
- Cryptographic Machine Instruction Formats
- Cryptographic Parameter List Reference
- Cryptographic Program Checks and Status Codes

VOLUME 6: CONTROL PROGRAM GENERATION (GC31-2071)
- Overall View of Control Program Geneneration (CPGEN)
- Sample CPGEN
- CPGEN Macro Statements (Reference)
- Using the Local Configuration Facility (LCF)
- CPGEN Messages
- LCF Messages

Figure 0-1. 4700 Controller Programming Library (GBOF-1387)

Contents

Figures

Chapter 1. 4700 Cryptography

Today's financial institutions face a serious challenge: How can data be made more secure as it moves through the institution's complex data processing system?

Data protection has always been a major concern, but recent trends have increased its importance:

- Greater use of electronic funds transfer

- More distribution of processing from a central computer site to remote locations

- Increased complexity of telecommunication networks

- Increased sophistication of electronic surveillance techniques

The hardware devices and programming tools provided by the IBM 4700 Finance Communication System can help your institution deal with this problem.

Two Examples of 4700 Cryptography

Consider as a hypothetical example the *Suburban-City Bank,* shown in Figure 1-1. This bank has one large downtown office and several branch offices in outlying parts of the city.

Suburban-City maintains its customer account information at its downtown central site. At each branch, an IBM 4701 Controller and attached terminals (IBM 4704 Displays and IBM 4710 Receipt/Validation Printers) aid tellers in handling transactions. The host computer and the branches are connected by telecommunication lines.

An Offline PIN Validation

Suburban-City Bank has given each customer a magnetically-encoded identification card and a personal identification number (PIN). When a customer begins a transaction, the teller passes the customer's identification card through the 4704 magnetic stripe unit, and then asks the customer to enter their personal identification number at the 4704's encrypting PIN keypad.

As the customer enters the PIN, the keypad automatically enciphers it in accordance with a key that Suburban-City has stored in the keypad. Thus the PIN, were it to be electronically intercepted before reaching the 4704 or its controller, would not be intelligible.

The Suburban-City program in the controller verifies that the PIN is valid, and notifies the teller to continue with the transaction.

The program can verify the PIN without communicating with the host computer, without deciphering the PIN, and without comparing the PIN with a stored list of PINs.

Encrypted Message Transmissions

Each time a Suburban-City teller processes a transaction that affects customer account information, Suburban-City sends the transaction to the downtown office. The branch and the downtown office are connected via public telephone lines, so sensitive messages are exchanged in enciphered form.

When the Suburban-City program in the 4701 initiates a session with its counterpart in the host computer, it requests an enciphered session. The host program can comply with this request because Suburban-City Bank has installed a *cryptographic subsystem* at its central site. Suburban-City Bank uses another 4701 controller for this purpose, locally-attached to its host computer. (Other cryptographic subsystems are available, as described later.)

The 4701 application program at the branch office generates a key and exchanges it in enciphered form with the host program. Both programs then encipher the data being transmitted and decipher the data as it is received. The 4700 encryption facilities enable the resulting session to have these characteristics:

- Although the session keys are related to keys that Suburban-City Bank stores in the hardware, neither program has access to the hardware keys. The programs handle even the session keys indirectly, and cannot determine their values either.

- Each eight-byte block of enciphered data is related to the previous block in a way that only the two programs can determine.

- The two programs can detect when an alteration has been made to the enciphered data, or when one enciphered message has been substituted for another.

- By systematically altering each successive transmission, the two programs can detect attempts to insert a duplicate message into the session.

By employing these techniques, Suburban-City Bank minimizes the likelihood that its daily transactions will be intercepted and misused.

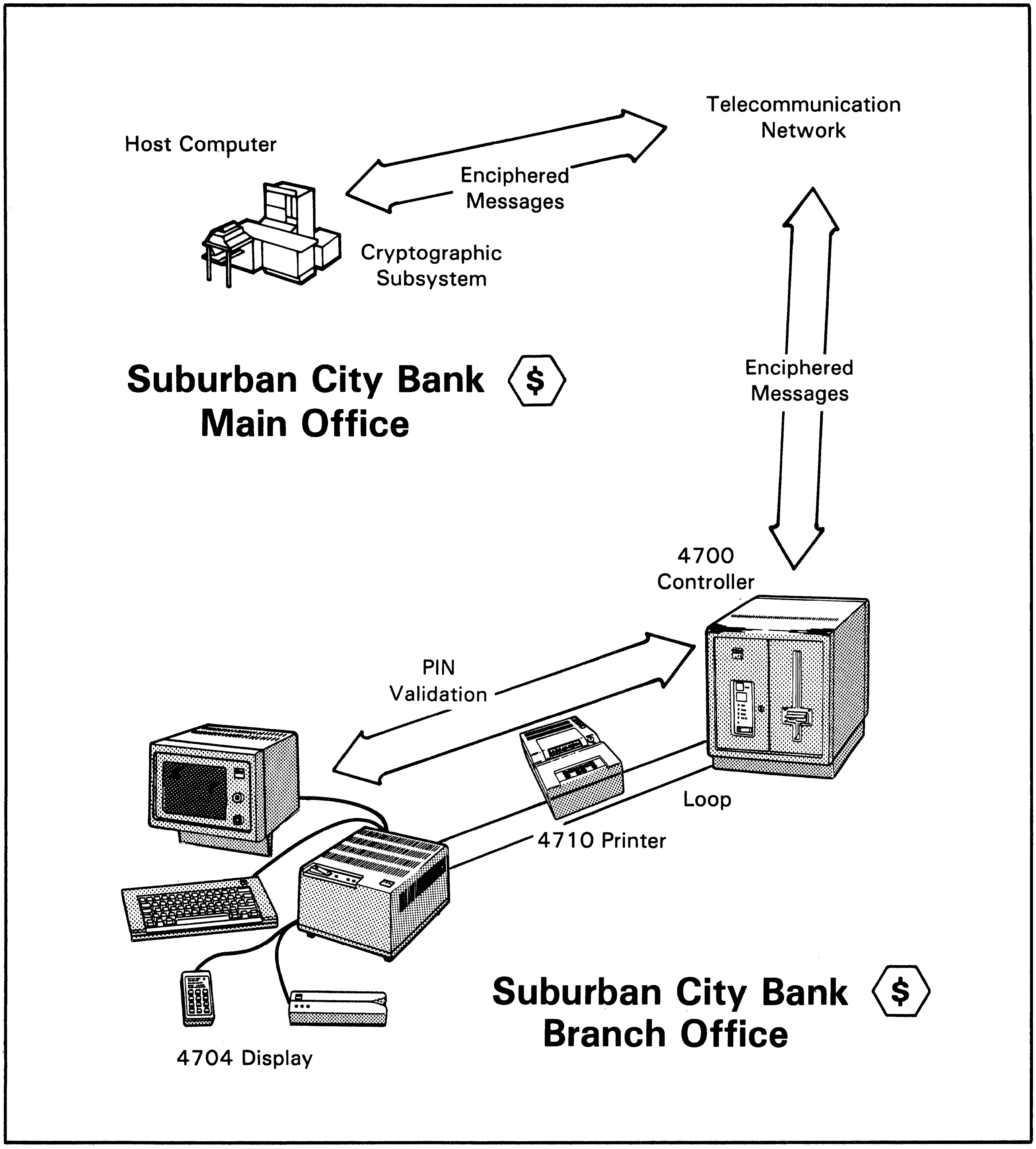


Figure 1-1. Cryptography in a Banking Environment

Who Uses the 4700 Cryptographic Facilities?

This book contains information for two kinds of users:

- *Programmers* who write programs that run in the operational system. Typical programmer tasks include enciphering data, exchanging keys with a program in the host computer, and validating PINs.

- *Security specialists* who plan, install, and manage the system in which the operational programs run. Typical security specialist tasks include generating, enciphering, and loading keys.

The information needed to perform these tasks is extensively interrelated. The person responsible for generating and enciphering keys can do so with the same tools available to an operational program -- that is, by writing and executing a utility program in the 4701 controller to aid in installing or altering the system. Conversely, the person responsible for writing the operational program needs to be aware of the manner in which the cryptographic key data sets are defined, and so forth.

After you have become familiar with the 4700 cryptographic facilities, you will see that they are designed so you can largely insulate the *security specialist* functions from the *programmer* functions. Although smaller institutions may elect to have one person be responsible for both jobs, it is better if you keep the two separate. This way the information generated by the security specialists can be maintained in a confidential manner.

Throughout this book, "you" generally means any person installing and using the cryptographic facilities. Where a distinction needs to be made between the person installing the system and the person writing the program, the latter is simply referred to as "the program" -- for example, "You must generate the enciphered keys and make them available to the program."

In general, programmers will use only the facilities described in Chapter 10, “4700 Cryptographic Instructions.”

Security specialists have the additional responsibilities listed below. If you are new to 4700 cryptography, many of the terms used below will be unfamiliar. These terms are explained in the next five chapters.

- Install the supporting modules of controller data.

- Generate random master keys for the controller; determine the key’s verification code and its two-part loading sequence. Arrange for these keys to be distributed to the controller (by courier, for example), loaded, and verified.

- Generate PIN keys for the encrypting PIN keypads; determine each key’s 24-keystroke loading sequence. Arrange for the key to be distributed to the keypad, loaded, and tested.

- Generate PINs, PIN validation and offset data, PIN protection keys, and PIN validation keys.

- Generate one sending and one receiving cross-domain key (optional) to be maintained in controller storage. Determine the verification code and the one-part (ciphertext) or two-part (plaintext) loading sequence for these keys.

- Generate variants of the master keys. Encipher cross-domain keys and PIN protection keys under the variants, and make the resulting cryptograms available to the using 4700 program.

- When it becomes necessary to change the controller's master key, generate new variants and encipher the cross-domain keys and PIN protection keys under them.

The security specialist should always perform these actions in as controlled and secure an environment as possible.

Installing the Supporting Modules of Controller Data

4700 controller functions are supported by modules of controller data. These modules are supplied by IBM on the installation diskette and in the distribution tape reel that accompanies the 4700 Host Support program (5668-989).

Two modules support 4700 cryptographic facilities

- P57 -- supports 3600-level cryptography

- P28 -- supports 4700-level cryptography

You specify the module(s) you want during the controller configuration procedure by including the operand *P57* and/or *P28* on the OPTMOD configuration statement. (This can be done at the host system, using the 4700 Host Support program, or it can be done at the controller site, using the 4700 Local Configuration Facility.)

Module P57 requires 500 bytes of controller storage for controller data; Module P28 requires 4000 bytes. If you install either of these modules, an additional 500 bytes is required for configuration data (if you install both, the total is still 500 bytes).

Neither module depends on the other, so you can save controller storage by omitting the module you don't need. If you want *no* encryption support, you can omit both modules. If you omit them, the controller operates normally, but returns an error response if the program or operator attempts to use a cryptographic facility.

During program execution, any program can interrogate Segment 15 to determine if the P57 or P28 modules are present in the controller. The COPY DEFGMS instruction (described in *Volume 1: 4700 General Controller Programming)* provides the displacements and values for the segment 15 indicators:

Field	Value	Meaning
GMSFTR	GMSFDEM	P57 Module present (X'0100' bit on)
GMSFT2	GMSF28M	P28 Module present (X'0800' bit on)

3600-Level Cryptography

The P57 module supports two controller instructions -- ENCODE and DECODE -- and emulates the DES cryptographic support available for the 3600 Finance Communication System. This support is comparable to *electronic code book* cryptography, as defined in American National Standards Institute publications.

If you have an existing 3600 controller application program and simply wish to migrate it for use in 4700 controllers, use 3600-level cryptography (Chapter 9, "Maintaining Compatibility Between Different Levels of Cryptography").

3600-level cryptography lacks several of the advantages of 4700-level cryptography (Figure 1-2). Among the most significant is the fact that the program using 3600-level cryptography handles all keys in plaintext. Such "keys-in-the-clear" cryptography is vulnerable to anyone who can gain access to program storage.

You need not use 3600-level cryptography to exchange enciphered data with 3600 controllers and host computers that employ this level of cryptography, - in fact, such use is not recommended. There is also no need to use 3600-level cryptography to communicate with a 3624 consumer transaction facility. Although 3600-level cryptography would work in these circumstances, you would not have access to the enhanced cryptographic functions described below, and you would have to handle keys in plaintext.

Although use of 3600-level cryptography in an operational environment (that is, by programs that are implementing your financial applications) is not as secure as 4700-level cryptography, you may wish to use 3600-level cryptography when you are *installing* your cryptographic system. In particular, the ENCODE instruction can be useful in building cryptographic key data sets, because ENCODE allows you to encipher under any key. Although *operational programs* should not use plaintext keys, security personnel can use them in a secure environment.

3600—Level Cryptography (P57)	4700—Level Cryptography (P28)
• Plain keys • Eight bytes per ENCODE or DECODE instruction • Electronic Code Book cryptography	• Enciphered keys • Up to 4096 bytes per instruction, with cipher block chaining • Electronic Code Book cryptography (with enciphered keys) available • Physical keylock • Master key and two cross-domain keys in cryptographic storage • Operator controlled facilities for key management and testing • Additional program-controlled instructions for managing keys, PINs, and message authentication codes

Figure 1-2. Levels of Cryptography

4700-Level Cryptography

The 4700 system provides the following cryptographic facilities not available on 3600 systems:

- A P28 module of controller data
- A controller with cryptographic storage and a physical encryption keylock (controls access to the cryptographic storage)
- System monitor and installation diskette procedures that you can use to load, verify, erase, and test the cryptographic facilities
- An encrypting PIN keypad, an accessory of the IBM 4704 Display

The P28 module provides support for:

- Enciphering and deciphering text with cipher block chaining
- Generating keys
- Deciphering keys and reenciphering them under a new key, without making the deciphered key available in plaintext
- Validating enciphered PINs, without making the deciphered PIN available in plaintext
- Translating enciphered PINs from one format to another, without making the deciphered PIN available in plaintext

- Reenciphering PINs from under one key to under another, without making the deciphered PIN available in plaintext

- Generating message authentication codes

These facilities are described in detail in subsequent sections of this book.

Cryptography at the Host Computer

The 4700 controller application program can perform several cryptographic functions offline, such as the PIN validation described earlier. Other functions require a cryptographic facility at the host computer or elsewhere in the data processing network.

Several facilities that implement the data encryption algorithm are available for use in or with host computers. These facilities are listed in Figure 1-3. Note that some use an equivalent of 3600-level cryptography, while others use an equivalent of 4700-level cryptography.

You can obtain a greater degree of security if you install 4700-level cryptography at your host computer. The two ways to do this are:

- If your host computer uses OS/VS (OS/VS1 or OS/VS2-MVS), install a 3848 Cryptographic Unit, or its software counterpart, the Programmed Cryptographic Facility. Chapter 11, "Guidelines for Coordinating 4700 and OS/VS Cryptographic Facilities" tells you how to equate each 4700 cryptographic facility with its counterpart in these OS/VS subsystems.

- If you can connect a 4700 controller to your host computer in the same room (where you can control access to them), you can use the 4700 controller itself as a cryptographic subsystem. Your program in this locally-attached controller can serve two purposes: you can use it to generate keys and other sensitive information needed to set up your system, and your operational programs in the host computer can use it to process cryptographic requests.

The *3848 Cryptographic Unit* is a channel-attached device. It requires the Cryptographic Unit Support program (5740-XY6). This program runs under OS/VS.

The *Programmed Cryptographic Facility* (5740-XY5) is a software equivalent of the 3848. The Programmed Cryptographic Facility also runs under OS/VS.

BDKDES is a 3600-level cryptographic routine that can be ordered with 4700's Host Support program (5668-989). Host Support runs under both OS/VS and VSE (DOS/VS). OS/VS users must order feature code 1611 (1600 bpi) or 1612 (6250 bpi) to obtain this feature; VSE users must order feature code 1609 (1600 bpi) or 1610 (1600 bpi). BDKDES is shipped separately from Host Support, and must be link-edited into Host Support before it can be used. BDKDES is described further in Chapter 8, "Host Support Encryption Routines."

The System/34 *Finance Subsystem* runs under the System/34 System Support program (5726-SS1) and requires the Interactive Communications Feature. The subsystem provides 3600-level cryptography for its using programs. See *IBM System/34 Interactive Communications Feature Reference Manual,* GC21-7751, for more information.

An IBM-supplied program is available that exchanges data between the System/34 computer and terminals attached to the controller. This program, the 4700 Online Terminal Support for System/34 (5799-BGB) supports encryption, but only between the controller and attached 3624 consumer transaction facilities. See *IBM 4700 Online Terminal Support for System/34 Programming Description and Operation,* SC31-0023, for more information.

Host System	Cryptographic Facility	Cryptographic Level*
Any system with 4700 controller in same (secure) room	4700 Controller	4700
OS/VS	3848 Cryptographic Unit	4700
OS/VS	Programmed Cryptographic Facility	4700
OS/VS	BDKDES	3600
VSE	BDKDES	3600
System/34	Finance Subsystem	3600

*4700-level = cipher block chaining with enciphered keys;
3600 level = plaintext keys, no cipher block chaining

Figure 1-3. Cryptographic Facilities for Host Computers

Chapter 2. Fundamentals of Enciphering and Deciphering

4700 cryptography is based on two fundamental operations -- *encipher* and *decipher:*

ENCIPHER

```
              +------------+
       Key ==>|    Data    |
 Plaintext ==>| Encryption | ==> Ciphertext
              | Algorithm  |
              +------------+
```

DECIPHER

```
              +------------+
       Key ==>|    Data    |
Ciphertext ==>| Encryption | ==> Plaintext
              | Algorithm  |
              +------------+
```

When the key and the text have been processed by the data encryption algorithm, the resulting ciphertext (enciphered text) can be returned to its original state only if the original key is available.

The key is an eight-byte value. The low-order bit of each byte is, in general ignored; you should choose the remaining bits in as random a manner as possible. (Key generation is explained further in Chapter 3, “Cryptographic Keys.”)

The text can be any data, even another key.

The algorithm used by the 4700 system is the *data encryption algorithm,* adopted by the United States National Bureau of Standards for use by Federal agencies requiring cryptographic protection of sensitive data. This algorithm is described in *Federal Information Processing Standard 46* (FIPS PUB 46), 15 January 1977, and in American National Standards Institute (ANSI) Standard X3.92. The 4700 controller implements this algorithm with cipher block chaining, as defined by *Federal Information Processing Standard 81* (FIPS PUB 81) and described further below.

Enciphered Data, Keys, and PINs

As you use the 4700 cryptographic facilities, you will often find it necessary to distinguish among three types of ciphertext: enciphered data, enciphered keys, and enciphered PINs.

Enciphered data is the enciphered form of the *message* you want protected. Enciphered data can be processed in single blocks as long as 4096 bytes.

Sometimes the text being enciphered is not data in the ordinary sense, but a *key.* Because keys are always eight bytes long, enciphered keys are also eight bytes long. A *cryptogram* is another term for an enciphered key.

The 4700 program handles each *PIN* in much the same manner as a key (an enciphered PIN is always eight bytes long), but the program does not itself encipher or decipher the PIN.

Notation

In this book, enciphered data is represented as

eK(Data)

where K is the key used to encipher the data. For example, eKS(Data) means "Data, enciphered under session key KS."

Enciphered keys are similarly represented as

eK1(K2)

where K1 is the key used to encipher K2. For example, eKM(KS) means "session key KS, enciphered under the master key, KM."

Enciphered PINs are represented as

eK1(PIN)

where K1 is one of the PIN protection keys and PIN is the formatted PIN.

Cipher Blocks

You can encipher or decipher up to 4096 bytes of data with one instruction. However, it is important to remember that the data encryption algorithm always processes text in eight-byte blocks.

If you submit a data string for encryption that is not a multiple of eight bytes, the controller pads the data as specified in your request.

Cipher Block Chaining

Repetitive patterns in multiple blocks of plaintext (unenciphered text) can result in repetitive patterns in the ciphertext. To make the ciphertext more resistant to analysis, the 4700 controller uses *cipher block chaining.*

The cipher block chaining technique works as follows (Figure 2-1).

When the controller has completed the enciphering of a given eight-byte block, it uses the enciphered data to modify the *next* block's plaintext. (The modification is an exclusive-OR operation on the two blocks.) Then the controller enciphers the next block's modified plaintext.

The controller repeats this process for each successive block.

The process is reversed when data is being deciphered. After the controller has deciphered a given eight-byte block, it uses the previous eight bytes of ciphertext to recover the original plaintext.

Note that the first block of data presents a special case. Because there *is* no previous block for the controller to use, you must supply one. You do this by including an *initial chaining value* (ICV) with the encipher (or decipher) request. The initial chaining value can be any eight-byte value.

If a portion of the enciphered data is altered during transmission, cipher block chaining allows only two blocks of enciphered data to be affected -- the block in which the error occurred, and the next block.

If you are deciphering data that was enciphered via cipher block chaining, you must know both the key *and* the initial chaining value that was used when the data was enciphered. If you do not have the initial chaining value available, you cannot recover the first eight bytes correctly.

A technique is described later in this book ("Example 4. Initiating a Cryptographic Session" on page 4-8) that permits both the sending and the receiving program (that is, the enciphering and the deciphering program) to keep track of the proper initial chaining values.

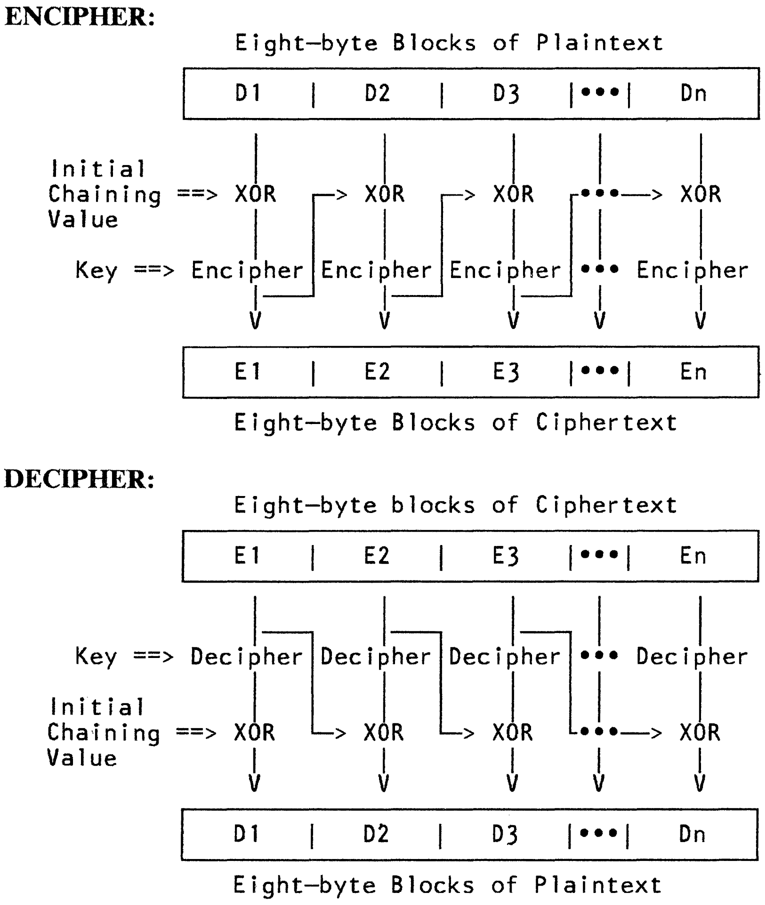


Figure 2-1. Cipher Block Chaining

Notes on Using ENCIPHER and DECIPHER

When the enciphered data is to be processed at another location (data exchanged via communication or diskette), the following must occur:

- The data encryption key (in this context, a "session key") must be available at the other location.

- The program at the other location must be capable of 4700-level cryptography -- that is, have access to the data encryption algorithm and be able to implement cipher block chaining. If the program has access only to 3600-level cryptography, the program must emulate cipher block chaining and provide pad processing.

- The initial chaining value must be known at both locations. You can provide this by using a fixed initial chaining value, by concatenating the initial chaining value with the enciphered data, or by using a synchronized sequence of initial chaining values. These methods are illustrated in Chapter 4, “Exchanging Keys with Programs in Other Domains.”

Chapter 3. Cryptographic Keys

The 4700 encryption facilities employ a wide range of keys for enciphering data, exchanging ciphertext with other programs, processing PINs, and authenticating messages.

The previous section showed that you use a key to encipher and decipher data, and mentioned that this key is enciphered under a *master key.* This section takes a closer look at these keys, and explains how they are structured and how they are generated and loaded. This section also introduces you to the other keys used by the 4700 cryptographic facilities.

The Controller Master Key (KM)

Each 4700 controller maintains one master key in cryptographic storage. The key is not accessible to programs.

You must install the master key (KM) before application programs can generate their own keys. You can use a "plaintext" version of the KEYGEN instruction to install KM; programs can then later use an "enciphered" version of the same instruction to obtain their data-encrypting keys.

The master key is never transmitted to the controller (downloaded), but must be entered manually at the controller. The master key is entered in two parts. This allows you to divide information about the key between two persons, so that neither has any awareness of the key -- this is sometimes referred to as a "dual-courier technique."

Before the master key is brought to the controller, you must:

- Generate the master key and its two parts
- Determine the key's *verification code.*
- Make the two parts and the verification code available at the 4700 controller.

The person or persons actually loading the key would:

- Start the controller.
- Activate the controller's encryption keylock with a physical key.
- Logon to the system monitor.
- Enter the key in two parts, using the 330-2-1 command.
- Use the 330-3-1 command to verify that the key has been correctly loaded (the command returns the verification code, which should match the code you generated).

(*4701 Controller Operating Procedures* tells you how to operate the controller, and *4700 Subsystem Operating Procedures* tells you how to use the system monitor.)

The overall process looks like this:

```
Generate  Form Two Parts  Load Plaintext Keys
========  ==============  ===================

  KM ────> KMa ─────┐                              4700 Controller
       └─> KMb ────>│                             ┌─────────────────┐
                    └───> 330-2-1 ──────────────>│ KM              │
                                                  │                 │
                                                  └─────────────────┘
```

Generating the Master Key

The 4700 KEYGEN instruction generates keys that are suitable for use as master keys. Or, you could use the key-generation facilities of the OS/VS cryptographic subsystem (described in Chapter 11, "Guidelines for Coordinating 4700 and OS/VS Cryptographic Facilities"). Like all keys used by the 4700 cryptographic facilities, a master key must have these characteristics:

- Contain 56 random bits
- Be eight bytes in length

Generating the Master Key Parts

To generate the two parts of the controller master key,

1. First generate KM itself -- using one of the methods noted above.

2. Generate a second key in a similar manner, *and treat it as the second key part.*

3. Exclusive-OR the two keys (the master key and the second key part), *and treat the result as the first key part.*

This process is illustrated below, where KM represents the master key, and KMa and KMb represent the two key parts.

```
In a secure environment:

   ==> KEYGEN ==> KM

   ==> KEYGEN ==> KMb

   KM
   KMb
   ==> XOR ==> KMa
-------------------------------------------------------
Later, at the branch controller:

   KMa
   KMb
   ==> 330-2-1 ==> KM (in cryptographic storage)
```

Determining the Verification Code

After you have loaded a key, the controller never displays that key to anyone. To verify that the key in the controller is in fact the key you think is there, you can ask the controller (via the 330-3 command) to display the key's verification code. This code is a two-byte "shorthand" version of the real key. If this code matches the one you obtained after you first installed the key, the key in the controller is probably the correct one.

The controller always displays the verification code when you first load the master key into the controller. So the easiest way to initially obtain the verification code is to load the master key and note the resulting verification code.

You do not have to do this using the controller that will actually use the master key. A master key that is loaded into two different controllers will produce the same verification code. You can use one central controller to generate keys and verification codes for all your branch controllers.

If you want to determine a key's verification code without actually loading the key, you can use this procedure: Encipher the master key with the 3600-level ENCODE instruction, where the master key serves as both the *data* and as the *key*. The last two bytes of the resulting eight bytes of ciphertext form the verification code.

You can also use an OS/VS cryptographic subsystem to determine the master key's verification code. To do this, first use the OS/VS EMK macro instruction to encipher the controller master key under the *host* master key (KMH), then use the OS/VS CIPHER macro instruction to reencipher the controller master key under itself.

Master Key Variants

Variants of the master key allow the controller to act as though several master keys are in use, when in fact there is only one master key.

As noted above, session keys and message authentication keys are enciphered under the master key itself. However, other keys are enciphered under master key *variants*. This scheme serves to isolate a program's access to a given cryptographic facility.

A variant results when the controller (or you) modifies the master key in accordance with a simple algorithm described later in this section.

Before running your operational programs, you encipher the keys that you wish to protect under the variant. You make the enciphered keys available to your operational programs -- for example, by placing the enciphered keys in diskette data sets.

The controller uses three variants of the master key -- KM1, KM2, and KM3.

(At this point, you may wish to unfold Figure F-1 on page X-6 from the back of this book and refer to it as we discuss the various keys. This figure summarizes all the keys used by the 4700 cryptographic facilities.) Each variant is associated with specific types of keys, as shown in the following table. This table includes the master key itself for comparison. (The master key can be considered as variant zero.)

Master Key Variant	Used to Encipher
KM0 (Master Key)	• Session keys (KS) • Message authentication keys (KMAC)
KM1 (1st Variant)	• Sending cross-domain keys (KCD1, KCDs) • Output PIN protection keys (KP2)
KM2 (2nd Variant)	• Receiving cross-domain keys (KCD2, KCDr)
KM3 (3rd Variant)	• Input PIN protection keys (KP1) • PIN validation keys (KPv)

First Variant (KM1)

You use *KM1*, the first variant of the master key, to encipher sending cross-domain keys and output PIN protection keys (these keys are explained below).

One method of loading a sending cross-domain key into the controller requires that you first encipher the key under the first variant of the master key.

The RFMK instruction (described below) transforms a session key from encryption under the master key to encryption under a sending cross-domain key.

One form of RFMK requires the program to provide the sending cross-domain key enciphered under the first variant.

Similarly, one form of the PINTRANS instruction transforms a PIN from encryption under one key to encryption under an output PIN protection key. PINTRANS requires the program to provide the output PIN protection key enciphered under the first variant.

Second Variant (KM2)

You use *KM2*, the second variant of the master key, to encipher receiving cross-domain keys.

If you are loading a receiving cross-domain key into the controller in enciphered form, you must first encipher that cross-domain key under the second variant of the master key.

The RTMK instruction transforms a session key from encryption under a receiving cross-domain key to encryption under a master key. One form of RTMK requires the program to provide the receiving cross-domain key enciphered under the second variant.

Third Variant (KM3)

You use *KM3*, the third variant of the master key, to encipher input PIN protection keys and PIN validation keys. The PINTRANS instruction requires the program to provide the input PIN protection key enciphered under the third variant. The PINVERIF instruction requires the program to encipher both the input PIN protection key and the PIN validation key under the third variant.

Generating Variants of the Master Key

The algorithms for converting master keys to master key variants are included in the controller's P28 module. You must perform the equivalent conversion (by hand or program) so that you can encipher the keys under the proper variant.

To create a master key variant, invert two bits in *each* of the key's eight bytes (you can do this by performing an exclusive-OR operation between each byte of the key and the value shown below):

Key	Bits to be Inverted	Value to XOR with Key
KM1	Third and seventh bits	X'22'
KM2	First and fifth bits	X'88'
KM3	Second and sixth bits	X'44'

Enciphering Under Variants of the Master Key

After you have determined the master key variant, you must encipher each key under the appropriate variant.

Enciphered Keys		Used By
eKM1(KCDs)	Sending cross-domain keys	RFMK
eKM1(KCD1)	Sending cross-domain keys	RFMK, 330-1-3 command
eKM1(KP2)	Output PIN protection keys	PINTRANS
eKM2(KCDr)	Receiving cross-domain keys	RTMK
eKM2(KCD2)	Receiving cross-domain keys	RTMK, 330-1-4 command
eKM3(KP1)	Input PIN protection keys	PINTRANS, PINVERIF
eKM3(KPv)	PIN validation keys	PINVERIF

One method for enciphering a key under a given variant is to use the 3600-level ENCODE instruction, where you supply the master key variant as the *key* and the key being protected (such as KCDs or KP2) as the *data.*

Or, you can use OS/VS cryptographic subsystem facilities, as described in Chapter 11, “Guidelines for Coordinating 4700 and OS/VS Cryptographic Facilities.”

Cross-Domain Keys

You use cross-domain keys to protect data-encrypting keys being exchanged between two locations (nodes). Data-encrypting keys include session keys (keys that programs use to encipher and decipher data) and message authentication keys. This protection applies to keys that are being stored on some physical exchange medium (diskettes, for example) as well as keys that are being transmitted from one location to another.

If all locations possessed the master keys of all other locations, cross-domain keys would serve no purpose -- the master key would itself suffice. But far greater security is possible if the keys held in common by two or more locations are *not* master keys.

The 4700 cryptographic facilities (as well as OS/VS cryptographic subsystems) maintain two independent cross-domain keys: one for sending and one for receiving.

The 4700 cryptographic facilities maintain these cross-domain keys in two different ways, depending on whether you keep the keys in *cryptographic* storage or in *program* storage (unprotected storage).

You can maintain two cross-domain keys in the controller's cryptographic storage, and an unlimited number of enciphered cross-domain keys in program storage or on diskette data sets.

Cross-Domain Keys in Cryptographic Storage

The controller maintains *one* sending and *one* receiving cross-domain key in cryptographic storage. These keys are designated *KCD1* (sending) and *KCD2* (receiving). Like the master key itself, KCD1 and KCD2 are long-term keys you load into the controller; these keys remain in storage until you erase them or replace them with new ones. If the controller application program exchanges data-encrypting keys with a program in *only one other node* (typically, the host computer), and has no need to use additional pairs of cross-domain keys, KCD1 and KCD2 are the only cross-domain keys you need to install.

Enciphered Cross-Domain Keys in Program Storage

If a program must exchange data-encrypting keys with multiple locations, you must encipher the cross-domain keys under the appropriate variant and provide the enciphered keys to the using program. We designate the keys that you *don't* maintain in cryptographic storage (and must therefore protect under a variant) as *KCDs* and *KCDr*. You must encipher the sending cross-domain key (KCDs) under KM1, and the receiving cross-domain key (KCDr) under KM2 in the manner described above.

	Sending Cross–Domain Key	Receiving Cross–Domain Key
As maintained in Cryptographic Storage	KCD1	KCD2
As available in Program Storage	eKM1(KCDs)	eKM2(KCDr)

The program uses RFMK and RTMK instructions to reencipher the exchanged keys. When the program prepares to send a data-encrypting key to another program (a session key, for example), it uses the RFMK instruction to transform the key from encryption under the master key to encryption under the appropriate sending cross-domain key (KCD1 or KCDs).

The program at the remote location uses the RTMK instruction (or equivalent) to reencipher the session key to encryption under *its* master key. To the receiving program, this same cross-domain key is a *receiving* cross-domain key (KCD2 or KCDr).

Example 1: Exchanging Session Keys (Using Cross-Domain Keys in Cryptographic Storage)

The following example relates the use of cross-domain keys with the RFMK and RTMK instructions. In this example (Figure 3-1), a controller application program at Installation A is sending an enciphered message to another program at Installation B. The program at B is shown as another controller application program, although any host program using an OS/VS cryptographic subsystem would suffice.

The program at A communicates only with the program at B, so the only cross-domain keys used are those maintained in cryptographic storage: KCD1 and KCD2. (KCD1 and KCD2 would already have been loaded into the controller, as explained later in this chapter.)

1. Program A first issues a KEYGEN instruction to generate a session key (KS) that is enciphered under controller A's master key (KMa).

2. KS is already enciphered under A's master key, rendering it useless to Program B (Program B does not have access to KMa). Program A invokes an RFMK instruction, causing eKMa(Ks) to be reenciphered under A's *sending cross-domain key.* The cross-domain key is available to both programs, but neither A nor B can obtain the cross-domain key in plaintext.

3. After A transmits eKCD1(KS), B receives it and invokes an RTMK instruction; this causes KS to be reenciphered under B's master key, KMb.

4. Program B now has KS in its proper form for the DECIPHER instruction: eKMb(KS). As A enciphers data under KS and transmits it to B, B can issue DECIPHER instructions and obtain the data in plaintext.

Note that when Program A reenciphered the session key (Step 2), it reenciphered it under the key it referred to as A's *sending* cross-domain key; whereas Program B reenciphered it under a key it referred to as B's *receiving* cross-domain key. Both keys are, however, the same key.

```
     Installation A                     Installation B
     ==============                     ==============

1.   ==> KEYGEN ==> eKMa(KS)

2.   eKMa(KS)
     (KCD1 in A's cryptographic
      storage)                        3.
     ==> RFMK ==> eKCD1(KS) ===========> eKCD2(KS)
                                         (KCD2 in B's cryptographic
                                          storage)
                                         ==> RTMK ==> eKMb(KS)
     Data
     eKMa(KS)                         4.
     ==> ENCIPHER ==> eKS(Data) =======> eKS(Data)
                                         eKMb(KS)
                                         ==> DECIPHER ==> Data

              Note:  A Sending         B Receiving
                       KCD1 <---same     ....
                       ....     key---> KCD2

                     A Receiving       B Sending
                       ....     same---> KCD1
                       KCD2 <---key      ....

                     A's sending key is therefore
                     B's receiving key, and vice-versa
```

Figure 3-1. Exchanging Session Keys

Suggestions for Using Cross-Domain Keys

You can define multiple sets of cross-domain keys. The number you define depends on the type and number of key exchanges you want to perform. This enables you to exchange keys in a variety of ways. For example, a program that exchanges keys with multiple programs can use the same sending and receiving pair for all of the programs, or the program can use a unique pair for each.

Unique cross-domain keys can also be defined for each type of key you are exchanging. You might define one cross-domain key for the purpose of exchanging a message authentication key, and a different cross-domain key for the purpose of exchanging encryption keys for files.

It is not necessary for every program in the network to have the same number of cross-domain keys. Nor is it required that the cross-domain keys be maintained in pairs.

A program that only receives keys from one source can have a single receiving cross-domain key and no sending key. A program that performs more complex key exchanges might define sending cross-domain keys for the keys it only sends, receiving cross-domain keys for the keys it only receives, and both sending and receiving keys for keys that it may have to both send and receive. (However, a single cross-domain key should not be used both as a sending and as a receiving cross-domain key.)

This flexibility also provides you with the ability to isolate certain functions within the network.

For example, suppose two programs communicate through an intermediate program. The data being sent is enciphered under a session key shared by the two communicating programs, but *not* known to the intermediate program. Also assume that this session key is periodically changed by one of the programs. The program that changes the key can send a new key to the other program, protected by a cross-domain key shared by the two programs. If the intermediate program also had access to the cross-domain key, then it could intercept that session key and use it to decipher the messages it is relaying.

To eliminate this possibility and ensure that the data and keys are available only to the appropriate program, do not allow the cross-domain keys to be known by the intermediate program. If the intermediate program requires a cross-domain key to exchange keys with either of the two "end" programs, you should define a separate cross-domain key for that purpose.

Generating and Loading Cross-Domain Keys

You can generate cross-domain keys in the same manner that you generate master keys -- that is, by using the 4700 KEYGEN instruction or the key generation facility of an OS/VS cryptographic subsystem.

If you are installing *KCDs* and *KCDr* cross-domain keys, you must encipher the key under the appropriate variant and make only the result available to the using program:

- eKM1(KCDs)
- eKM2(KCDr)

If you are installing *KCD1* and *KCD2* cross-domain keys, you must load them into the controller's cryptographic storage. There are two ways you can do this:

- In two parts, using the system monitor's 330-2 command

- In enciphered form, using the system monitor's 330-1 command (in this form, one person enters the entire key)

When you load a cross-domain key, the controller displays the two-byte verification code for the key. You can use this verification code in the same manner that you use master key verification codes (see "Determining the Verification Code" on page 3-3).

You load the sending and receiving cross-domain keys separately.

To load KCD1 or KCD2 in *two parts,* first generate the two parts in the same manner described earlier for loading master keys; then load the key parts with the system monitor's 330-2 command:

```
Generate  Form Two Parts  Load Plaintext Keys
========  ==============  ===================

  KCD1 ───> KCD1a ─────┐                        4700 Controller
        └─> KCD1b ────>│                      ┌──────────────────┐
                       │                      │ KM               │
                       └───> 330-2-3 ───────> │ KCD1             │
  KCD2 ───> KCD2a ────>┌───> 330-2-4 ───────> │ KCD2             │
        └─> KCD2b ─────┘                      └──────────────────┘
```

To load KCD1 or KCD2 in *enciphered form,* encipher the key under the appropriate variant, bring or transmit the result to the controller, then load the enciphered key with the system monitor's 330-1 command:

```
Generate   Encipher Key    Load Enciphered Keys
========  ==============  ====================

  KCD1 ──> eKM1(KCD1)──┐                        4700 Controller
                       │                      ┌──────────────────┐
                       │                      │ KM               │
                       └───> 330-1-3 ───────> │ KCD1             │
  KCD2 ──> eKM2(KCD2)──────> 330-1-4 ───────> │ KCD2             │
                                              └──────────────────┘
```

Data-Encrypting Keys and PIN-Encrypting Keys

You use the controller master key, master key variants, and cross-domain keys discussed above to encipher other keys. These key-encrypting keys are generated and installed before program execution.

In contrast, *data-encrypting keys* can be generated dynamically by the program. The program uses the KEYGEN instruction to generate the data-encrypting keys, which are always enciphered under the master key.

One of the most common uses for a data-encrypting key is to encipher and decipher data being exchanged between two programs at different locations. A data-encrypting key used for this purpose is the *session key (KS),* to which you were introduced in "Example 1: Exchanging Session Keys (Using Cross-Domain Keys in Cryptographic Storage)" on page 3-8.

Another common use for a data-encrypting key is to generate a message authentication code (MAC) as described in the next chapter. A data-encrypting key used for this purpose is a *message authentication key,* represented as *KMAC.*

In addition to the key-encrypting keys and the data-encrypting keys, the 4700 cryptographic facilities use a third group of keys exclusively for personal identification numbers (PINs). There are three keys in this group:

- KP1, an input PIN protection key
- KP2, an output PIN protection key
- KPv, a PIN validation key

You use KP1 and KP2 to encipher PINs, and KPv to generate a PIN validation code. These keys provide the same protection for PINs that key-encrypting keys provide for keys, and that data-encrypting keys provide for data.

Although these PIN-encrypting keys are similar to data-encrypting keys in some respects, you generate and install them in much the same manner as the key-encrypting keys discussed earlier. Because PIN-encrypting keys play a unique role in a 4700 cryptographic subsystem, they are described in detail in a separate chapter (see Chapter 6, "Validating and Translating Personal Identification Numbers").

Generating Data-Encrypting Keys

Session keys and message authentication keys can be generated dynamically by the program, or you can define and install them before program execution. Keys that you define prior to program execution must be stored in enciphered form -- under a remote location's master key or receiving cross-domain key, for example. Program-generated keys must be enciphered under the controller's master key; see the description of the KEYGEN instruction below.

Distributing Data-Encrypting Keys

You can distribute enciphered keys to the using controller on a diskette, transmit (download) them over telecommunication lines, or manually enter them at the controller by means of a user-supplied application program. Keys that are enciphered under cross-domain keys must be reenciphered under the controller's master key (RTMK instruction) before the program can use them.

You can distribute a dynamically-generated key to other programs in the network by transmitting the key enciphered under the remote location's receiving cross-domain key.

How Many Data-Encrypting Keys Does a Program Need?

You can define any number of data-encrypting keys, depending on the types of functions the program performs.

A program that communicates with only one other program may define a single session key and a single message authentication key. (Don't use the same key as a session key and as a message authentication key.)

More complex programs that communicate with programs in many different locations could use a unique pair of keys for each of the remote programs.

A program should generate a new session key and a new message authentication key each time it begins a session with another program. If the session extends longer than a day, then at least daily the program should stop the session, generate a new key, and then resume the exchange of ciphertext.

Generating Keys in a 4700 Program

The security provided by a program-generated key is proportional to the randomness of the key.

KEYGEN generates keys with a high degree of randomness. KEYGEN achieves this randomness by using frequently-changed areas of controller storage and by frequently and unpredictably using the controller clock.

4700 programs should use the KEYGEN instruction to generate their data-encrypting keys.

Chapter 4. Exchanging Keys with Programs in Other Domains

In the previous chapter, we saw briefly how a program can use two instructions to exchange a session key with another program:

RFMK Used when sending a data-encrypting key to another domain; converts the key from encryption under the master key to encryption under one of the sending cross-domain keys.

RTMK Used when receiving a data-encrypting key from another domain; converts the key from encryption under one of the receiving cross-domain keys to encryption under the master key.

Both instructions perform this conversion without providing the program access to its key in plaintext.

In this chapter, we will take a closer look at these instructions. We will also consider how a program might use them to overcome problems not evident in the simplified situation of Example 1.

The RFMK Instruction

The RFMK (Reencipher *From* Master Key) instruction takes a key that has been enciphered under the controller's master key (KM), deciphers it, and reenciphers it under a different key. The different key can be either of the two types of sending cross-domain keys:

- KCD1 (in cryptographic storage)
- eKM1(KCDs) (in program storage)

When you use RFMK to reencipher under *KCD1,* RFMK operates like this:

```
• eKM(K) ==> RFMK ==> • eKCD1(K)
```

Example 1 in the previous chapter used this version of RFMK.

When you use RFMK to reencipher under *KCDs,* RFMK operates like this:

```
• eKM(K)
• eKM1(KCDs) ==> RFMK ==> • eKCDs(K)
```

The next example in this chapter uses this version of RFMK.

Note that in both cases you start with K enciphered under one key and end up with the *same* key, now enciphered under a different key. In typical uses of RFMK, K is KS, a session key being exchanged between two programs.

The RTMK Instruction

The RTMK (Reencipher *To* Master Key) instruction takes a key that has been enciphered under one of the following keys --

- KCD2 (in cryptographic storage)
- eKM2(KCDr) (in program storage)

and reenciphers it under the controller master key, KM. When you use RTMK to reencipher a key that was enciphered under *KCD2,* RTMK operates like this:

```
• eKCD2(K) ==> RTMK ==> • eKM(K)
```

When you use RTMK to reencipher a key that was enciphered under *KCDr,* RTMK operates like this:

```
• eKCDr(K)
• eKM2(KCDr) ==> RTMK ==> • eKM(K)
```

Note that RTMK is essentially the inverse of RFMK, but the two instructions use different variants of the master key.

Three More Communication Examples

The following examples should help you answer these questions:

- How can a program exchange keys when more than two locations are involved? (See Example 2 below.)

- How can two programs verify that they are using the same session key? (See "Example 3. A Key Validation Protocol" on page 4-6.)

- What steps are involved in a session initiation procedure? (See "Example 4. Initiating a Cryptographic Session" on page 4-8.)

Example 2. Exchanging Session Keys (Using Cross-Domain Keys in Program Storage)

In Example 1, Program A transmitted a session key to B that was enciphered under a cross-domain key, KCD1. Because KCD1 is maintained in cryptographic storage, the program supplied only one input to the RFMK operation: eKM(K). In effect, the program requested the controller to reencipher K under the (only) sending cross-domain key available.

In Example 2, Program A can send session keys to programs other than Program B. Because the controller has only one pair of cross-domain keys in cryptographic storage, Installation A's security specialists have generated and stored additional pairs of cross-domain keys on a data set available to Program A.

To prevent these keys from being exposed in plaintext, the security specialists have enciphered the cross-domain keys under the appropriate master key variant. Three pairs of enciphered keys have been installed; one for communication with Installation C, another for communication with Installation D, and a third for communication with Installation F. (Program A can continue to use KCD1 and KCD2 to communicate with Program B, as in Example 1.)

Figure 4-1 (top) illustrates the enciphered keys that are available to Program A.

1. Program A issues the KEYGEN instruction and generates a session key, KS, enciphered under A's master key.

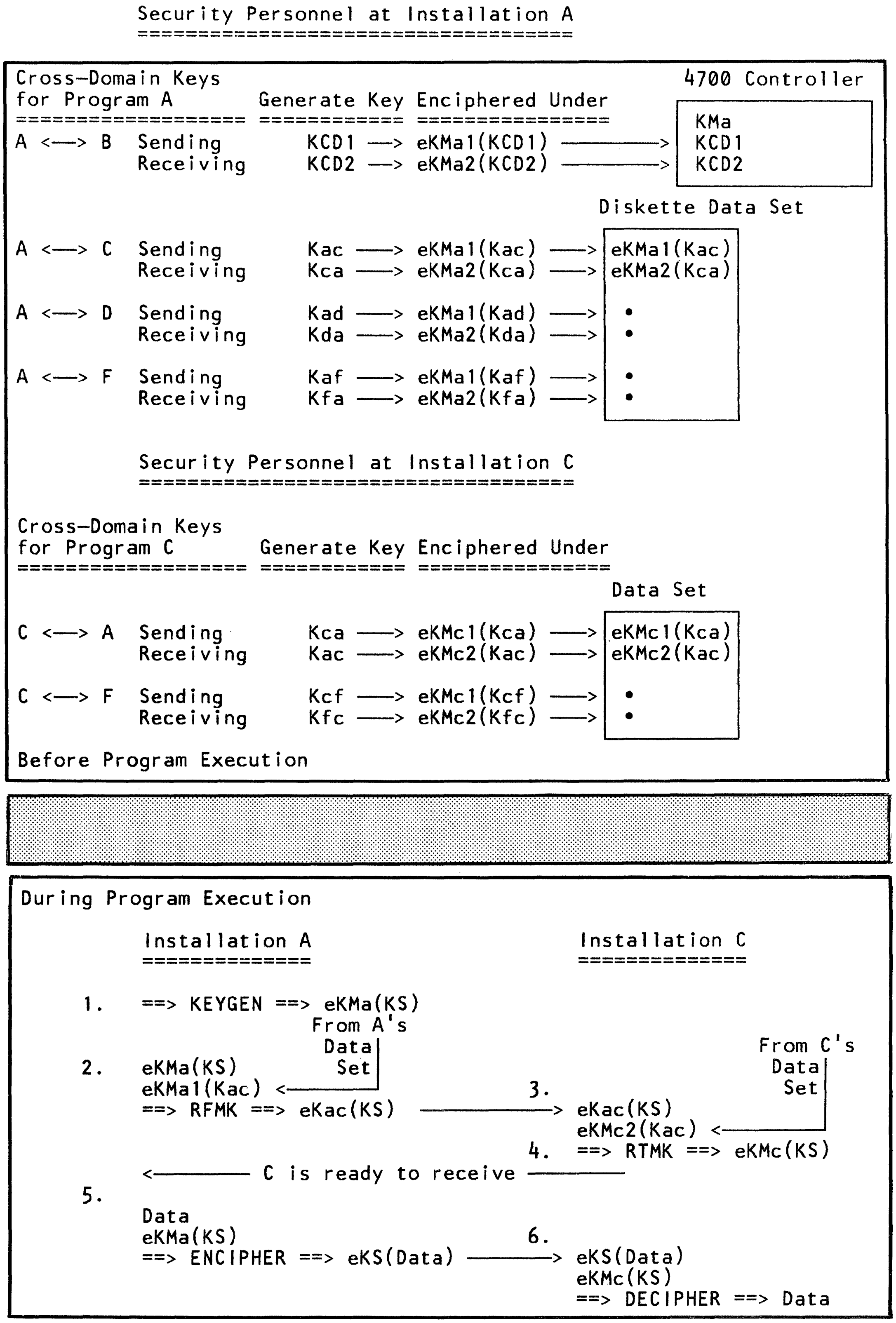
2. The program issues the RFMK instruction to reencipher the session key under the sending cross-domain key, Kac. (Kac is a specific example of the general class of sending cross-domain key, KCDs.) Note that the program need not (and in fact, cannot) access Kac in plaintext.

3. The program transmits the enciphered session key to Installation C.

4. At Installation C, the receiving program issues an RTMK instruction (or equivalent), using as input (1) the received enciphered session key, eKac(KS), and (2) the enciphered cross-domain key, eKMc2(Kac). Program C obtains the enciphered cross-domain key from a data set at Installation C. Note that Program C, like Program A, cannot obtain Kac in plaintext.

5. After Program C indicates that it is now ready to receive data from A enciphered under KS (more on this in Example 3, Step 8), Program A begins enciphering and transmitting the data.

6. Program C issues DECIPHER instructions (or their equivalent) and recovers the transmitted data in plaintext.



```
                    Security Personnel at Installation A
                    ====================================

Cross-Domain Keys                                                4700 Controller
for Program A         Generate Key Enciphered Under
================== =========== ================               KMa
A <--> B  Sending        KCD1 --> eKMa1(KCD1) ----------->  KCD1
          Receiving      KCD2 --> eKMa2(KCD2) ----------->  KCD2

                                                   Diskette Data Set

A <--> C  Sending        Kac ---> eKMa1(Kac) ---> eKMa1(Kac)
          Receiving      Kca ---> eKMa2(Kca) ---> eKMa2(Kca)

A <--> D  Sending        Kad ---> eKMa1(Kad) --->  •
          Receiving      Kda ---> eKMa2(Kda) --->  •

A <--> F  Sending        Kaf ---> eKMa1(Kaf) --->  •
          Receiving      Kfa ---> eKMa2(Kfa) --->  •

                    Security Personnel at Installation C
                    ====================================

Cross-Domain Keys
for Program C         Generate Key Enciphered Under
================== =========== ================
                                                     Data Set

C <--> A  Sending        Kca ---> eKMc1(Kca) ---> eKMc1(Kca)
          Receiving      Kac ---> eKMc2(Kac) ---> eKMc2(Kac)

C <--> F  Sending        Kcf ---> eKMc1(Kcf) --->  •
          Receiving      Kfc ---> eKMc2(Kfc) --->  •

Before Program Execution
```

```
During Program Execution

          Installation A                      Installation C
          ==============                      ==============

  1.      ==> KEYGEN ==> eKMa(KS)
                          From A's
                             Data                                 From C's
  2.      eKMa(KS)           Set                                     Data
          eKMa1(Kac) <--------               3.                      Set
          ==> RFMK ==> eKac(KS)   ------------> eKac(KS)
                                                eKMc2(Kac) <----------
                                             4. ==> RTMK ==> eKMc(KS)
          <--------- C is ready to receive ---------
  5.
          Data
          eKMa(KS)                           6.
          ==> ENCIPHER ==> eKS(Data) -------> eKS(Data)
                                              eKMc(KS)
                                              ==> DECIPHER ==> Data
```

Figure 4-1. Exchanging Session Keys (Using Cross-Domain Keys in Program Storage)

Example 3. A Key Validation Protocol

After accomplishing an exchange of keys, the programs in Examples 1 and 2 could continue sending data enciphered under KS, the session key.

But how can the programs be certain they are operating with the correct session key? The key might have been deliberately modified during transmission, or there might be a problem with the cryptographic data sets. Unless some mechanism is used to ensure correct reception of the session key, the two programs will continue to exchange unrecoverable data until they switch to a new session key.

To eliminate this problem, the programs should refrain from using KS for exchanging messages until they have *validated* the key.

The following example represents one possible key validation protocol.

In this example (Figure 4-2), the program at Installation A is sending an enciphered message to a program at Installation D. A's cryptographic data set is the same as that shown in Example 2 (Figure 4-1, top). D's cryptographic data set contains a pair of cross-domain keys for A, enciphered under D's master key: eKMd1(Kda) and eKMd2(Kad). (In the interest of simplicity, initial chaining values are not addressed until the next example.)

1. Program A generates an enciphered session key, eKMa(KS).

2. Using this enciphered key, Program A enciphers an arbitrary value N. The program at D does not have to know N ahead of time, but A and D must both share a common procedure for *altering* N. By limiting N to an eight-byte value, the programs can avoid cipher block chaining (Program A uses an initial chaining value of zero).

3. Program A reenciphers the session key under the proper sending cross-domain key, just as in the previous example.

4. Program A combines the results of the previous two steps, and sends this combination to Program D.

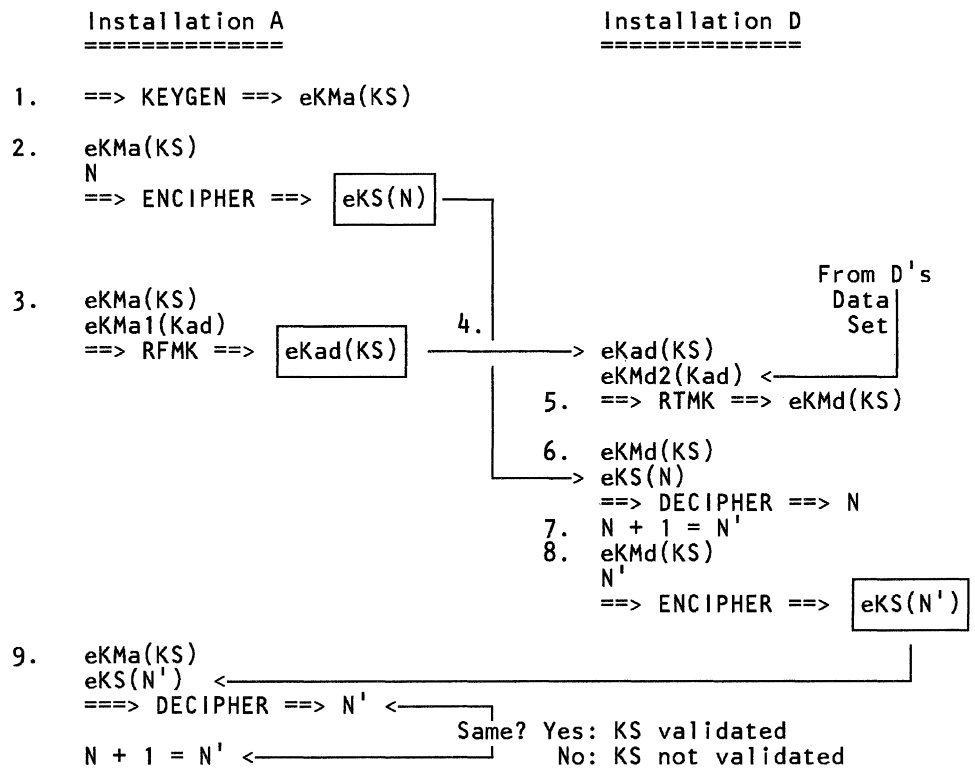
5. Upon receiving this message, Program D recovers KS by reenciphering KS from the cross-domain key to D's master key.

6. Program D deciphers the first part of the message, eKS(N), and obtains N in plaintext.

7. Program D next uses the common procedure for altering N. This procedure does not have to be complicated; any procedure that alters N in a predictable fashion is sufficient. For example, N' could be derived simply by incrementing N, as shown here.

8. Program D enciphers N' under KS and sends the results back to Program A as an acknowledgment message.

9. Program A deciphers the acknowledgment message and obtains N' in plaintext, generates its *own* version of N', and compares the two. If they compare, the key exchange has been successful, and Programs A and D can continue to communicate via KS. If they do not compare, key validation has failed.



```
     Installation A                          Installation D
     ==============                          ==============

1.   ==> KEYGEN ==> eKMa(KS)

2.   eKMa(KS)
     N
     ==> ENCIPHER ==> eKS(N)

                                                            From D's
3.   eKMa(KS)                                                   Data
     eKMa1(Kad)                  4.                              Set
     ==> RFMK ==> eKad(KS)            ------> eKad(KS)
                                              eKMd2(Kad) <
                                         5.   ==> RTMK ==> eKMd(KS)

                                         6.   eKMd(KS)
                                              eKS(N)
                                              ==> DECIPHER ==> N
                                         7.   N + 1 = N'
                                         8.   eKMd(KS)
                                              N'
                                              ==> ENCIPHER ==> eKS(N')

9.   eKMa(KS)
     eKS(N')  <
     ===> DECIPHER ==> N' <
                                    Same? Yes: KS validated
     N + 1 = N' <                          No: KS not validated
```

Figure 4-2. A Key Validation Protocol

There are a number of ways to handle a key validation failure.

Program A can attempt to send the key to Program D again, or it can generate a new key and send it to D. If the key exchange continues to fail, Program A could end its session with D and alert an operator that the programs cannot validate their key. In this instance, the operator or security specialist at A could reenter the cross-domain key at A or at D, and restart the system.

The security specialist may want to look for a potential security exposure that could have caused the problem. Or, Programs A and D could establish a backup procedure that allows the two programs to revert to an alternate cross-domain key. This should only be used as an emergency procedure however, because the inability to exchange keys in a secure manner adversely affects the overall security of the system.

Example 4. Initiating a Cryptographic Session

This example combines the key exchange technique described in the previous example with a technique for exchanging initial chaining values (ICVs). The result is a comprehensive session initiation protocol.

In this example, Program A initiates a session with a program at Installation F. The object is to exchange session keys and initial chaining values.

1. Program A builds the elements needed for a session initiation message:

   - a session key enciphered under the sending cross-domain key Kaf,

   - a check value, N, enciphered under the session key (with ICV=0), and

   - an initial chaining value, ICVfa, for use by F when sending to A.

     The actual value chosen for ICVfa is not important, but it should be as random as possible. In this example, A uses KEYGEN to generate ICVfa.

2. Program F receives the session initiation message and

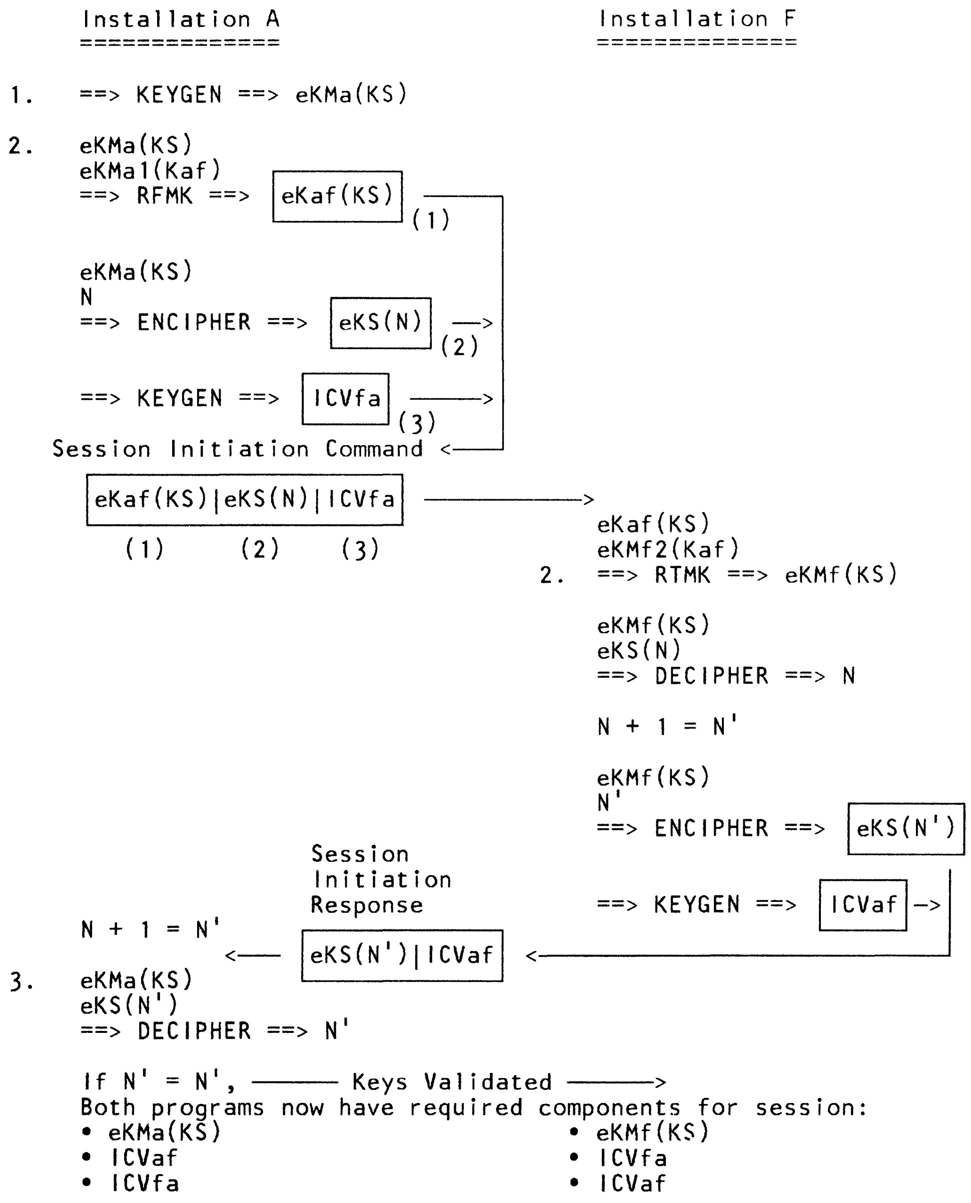
   - recovers the enciphered session key, eKMf(KS),

   - recovers N, generates N', and enciphers N' under KS (with ICV=0), and

   - generates an initial chaining value, ICVaf, for use by A when sending to F.

   Program F combines the enciphered N' and ICVaf into a response to A's session initiation command, and sends it to A.

3. Program A deciphers the enciphered N', generates its own version of N', and compares the two. If they match, Program A notifies Program F that the session has been successfully initiated. Each program now possesses both the enciphered session keys and initial chaining values needed for subsequent transmissions.



Chapter 5. Authenticating Messages

A 4700 controller application program can generate a *message authentication code* (MAC) that it uses to authenticate messages. Using this code, the controller program and a program at a remote location can detect whether a message has been altered during transmission. The program generates the code by issuing a MACGEN instruction.

The MACGEN instruction processes a data string (a message, for example) and returns a four-byte message authentication code. You provide MACGEN a key and an initial chaining value just as though you were enciphering the data.

Internally, the controller enciphers a copy of the data using cipher block chaining. Unlike ENCIPHER, however, the controller does not return the entire enciphered message to you, nor does it disturb the input data string.

MACGEN operates like this:

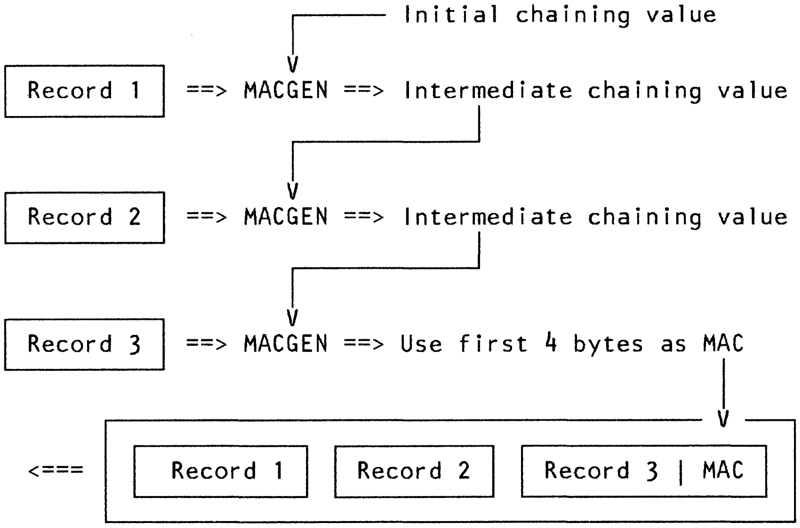
- Data
- eKM(KMAC) ==> MACGEN ==> MAC
- ICV
- [pad]

After the controller has enciphered the entire data string -- eKMAC(Data) -- it returns only the final eight bytes to you. You use only the *first four bytes* as the *message authentication code (MAC).*

The message authentication code is a function of the four inputs shown above. If any of these are changed, the code changes. If you were to generate a MAC at two different times using the same key, same initial chaining value, and same padding, yet the two MACs were not identical, you would know that the two data strings were not the same. Thus you can use the MAC to determine that the data string has been modified (a bogus message substituted for the original, for example).

You append a MAC to a message that you are going to store or transmit. If you receive or read a message that has a MAC appended to it, you generate your own MAC and determine if they match.

As noted above, MACGEN returns an eight-byte value, of which the first *four* are the MAC. You use the entire *eight* bytes in the manner described below when you wish to authenticate a message that is to be stored or transmitted in multiple physical records. You do not have to append a separate MAC for each record. You can instead invoke MACGEN once for each physical record, using the returned eight bytes as the *initial chaining value* for the *next* record's MACGEN. You would then append the final MAC to the last message of your transmission:


Initial chaining value
Record 1
==> MACGEN ==> Intermediate chaining value
Record 2
==> MACGEN ==> Intermediate chaining value
Record 3
==> MACGEN ==> Use first 4 bytes as MAC
<===
Record 1
Record 2
Record 3 | MAC

MACGEN Conventions that Both Programs Should Observe

To authenticate data for another program, the following must occur:

- The MAC must be included with, or appended to the data.
- The MACGEN key (KMAC) must be available to both programs.
- If the remote program is not a 4700 program, the remote program must have some means of performing an equivalent of MACGEN. A program that has access only to electronic code book cryptography would have to perform the same cipher block chaining functions (exclusive-OR, pad processing) as MACGEN.
- The initial chaining value must be known by both programs. You can do this by using a fixed initial chaining value, by including the initial chaining value with the data, or by using a synchronized sequence of initial chaining values for transmitting and receiving.

If the initial chaining value is fixed or is passed with the data, you may wish to take additional precautions. For example, you could include variable data within the data record and verify that the variable data is as expected when received.

During extended periods that KMAC remains unchanged, a successful MAC check does not mean that the message is timely; if the message was intercepted, delayed, and then replayed, the MAC still checks. To detect replayed messages, you have to examine the received data for some indication of currency. For example, you could check for a non-repeating sequence number or time-stamp that you can compare with a like value in program storage.

Authenticating Enciphered Data

In the above discussion, we assumed that a plaintext message was being authenticated. Message authentication works equally well, however, on enciphered messages.

If you are enciphering a message or any part of a message that you are also authenticating, you must create the ciphertext *before* you generate the message authentication code.

If you are using ENCIPHER to pad the data, and then using the same parameter list to generate an authenication code for the enciphered data; you may want to reset the pad request flag before issuing the MACGEN. (Remember the ENCIPHER with PAD incremented the length field to a multiple of eight bytes.)

Example 5. Authenticating Messages

In this example (Figure 5-1), Program A and Program F have already successfully initiated a session as shown in the previous example. Both programs possess the same session key and message authentication key (enciphered under their respective master key) and each has an initial chaining value to use for enciphering data being transmitted.

To detect any modification to the data being transmitted, both programs use the MACGEN (or equivalent) facility described above. Each program maintains a version of each other's initial chaining value (ICV); for each successive transmission, the programs modify the ICV in a fixed manner -- in this example, by incrementing the ICV.

1. Program A increments its sending ICV and enciphers the data.

2. Program A provides the incremented ICV, the message authentication key (KMAC), and the enciphered data being transmitted to the MACGEN instruction. Program A then appends the resulting message authentication code (MAC) to the enciphered data and sends the complete message to F.

3. Program F increments its receiving ICV (ICVaf) so that it remains identical to the ICV that A used for its MACGEN and ENCIPHER instructions. Program F separates the MAC from the ciphertext and issues MACGEN, repeating the MACGEN operation previously performed at A. If the resulting MAC is the same as the MAC appended to the data, the data has been authenticated and can be presumed to be unchanged. (If the MACs do not match, either the data or the MAC has been changed, or the two programs are not in synchronization. If the MAC check fails, Program F would request a new session.)

4. Program F deciphers the authenticated ciphertext.

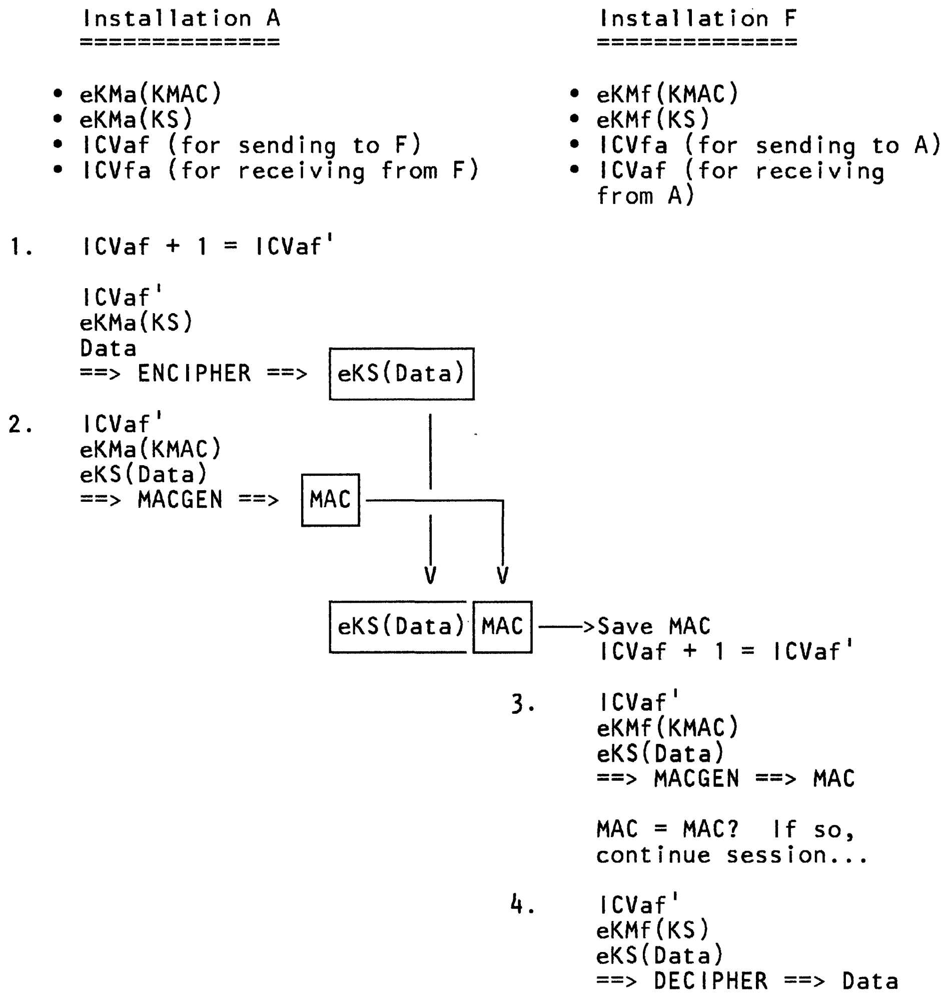
5. Program F now sends a message back to A, by performing steps similar to those used by Program A (Steps 1 and 2). Program F:

   - increments its sending ICV, ICVfa,
   - enciphers the data using the session key KS,
   - uses KMAC to generate a MAC for the ciphertext
   - appends the MAC,
   - and sends the message to A.

6. Program A authenticates the message just as Program F did (Steps 3 and 4) except that A uses its *receiving* initial chaining value, ICVfa. Program A:

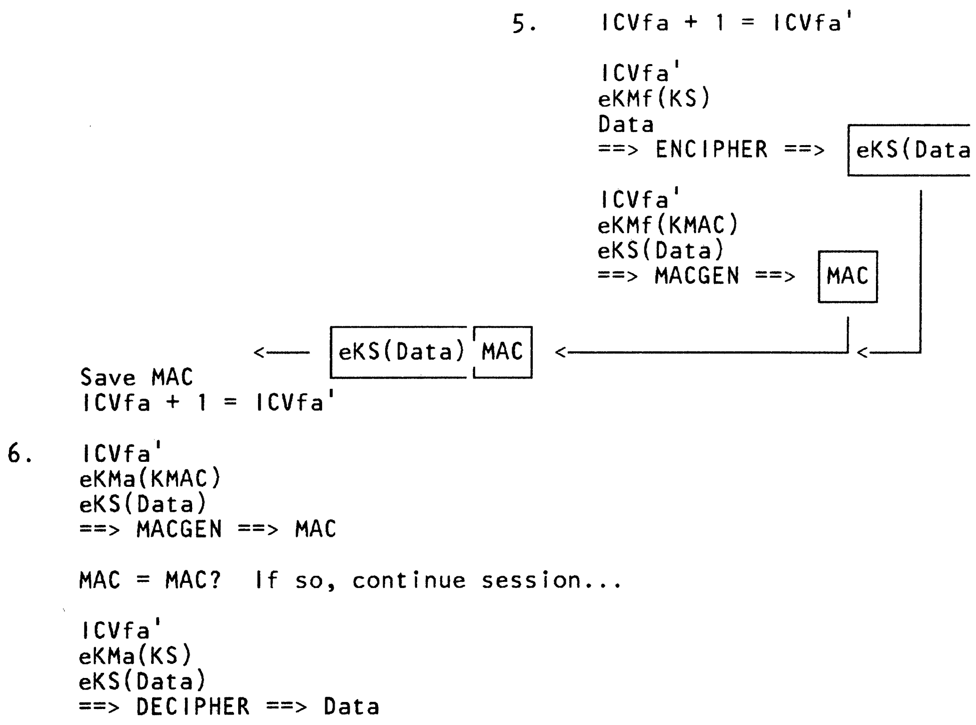
   - increments ICFfa,
   - generates a MAC,
   - verifies that the received MAC and the generated MAC match,
   - and deciphers the ciphertext.

Note that as the session continues, each program's version of ICVfa increments in unison. The same is true for ICVaf.



```
     Installation A                       Installation F
     ==============                       ==============

   • eKMa(KMAC)                         • eKMf(KMAC)
   • eKMa(KS)                           • eKMf(KS)
   • ICVaf (for sending to F)           • ICVfa (for sending to A)
   • ICVfa (for receiving from F)       • ICVaf (for receiving
                                          from A)

1.   ICVaf + 1 = ICVaf'

     ICVaf'
     eKMa(KS)
     Data
     ==> ENCIPHER ==> [eKS(Data)]
                            |
2.   ICVaf'                 |
     eKMa(KMAC)             |
     eKS(Data)              |
     ==> MACGEN ==> [MAC]---+----+
                            |    |
                            V    V
                    [eKS(Data)][MAC]--->Save MAC
                                        ICVaf + 1 = ICVaf'

                                  3.    ICVaf'
                                        eKMf(KMAC)
                                        eKS(Data)
                                        ==> MACGEN ==> MAC

                                        MAC = MAC?  If so,
                                        continue session...

                                  4.    ICVaf'
                                        eKMf(KS)
                                        eKS(Data)
                                        ==> DECIPHER ==> Data
```

Figure 5-1 (Part 1 of 2). Authenticating Messages



```
                                        5.    ICVfa + 1 = ICVfa'

                                              ICVfa'
                                              eKMf(KS)
                                              Data
                                              ==> ENCIPHER ==> eKS(Data

                                              ICVfa'
                                              eKMf(KMAC)
                                              eKS(Data)
                                              ==> MACGEN ==> MAC

                      <-- eKS(Data) MAC <--                      <--
      Save MAC
      ICVfa + 1 = ICVfa'

  6.  ICVfa'
      eKMa(KMAC)
      eKS(Data)
      ==> MACGEN ==> MAC

      MAC = MAC?  If so, continue session...

      ICVfa'
      eKMa(KS)
      eKS(Data)
      ==> DECIPHER ==> Data
```

Figure 5-1 (Part 2 of 2). Authenticating Messages

Chapter 6. Validating and Translating Personal Identification Numbers

The 4700 controller has two facilities that process personal identification numbers (PINs). A program using these facilities can:

- validate a PIN, if the program has access to PIN validation data, or
- translate a PIN, for use by a program at another location that has access to PIN validation data.

To invoke these facilities, the program issues a PINVERIF instruction (for PIN validation) and a PINTRANS instruction (for PIN translation). Both instructions require the P28 module in the 4700 controller.

PIN Validation

PIN validation is the process of comparing a customer-entered PIN with related data that you have defined and associated with the customer (placed on the customer's identification card, for example). When performed in the 4700 controller, PIN validation is an offline operation -- that is, no communication with the host computer is required.

The program uses a PINVERIF instruction to pass both the PIN and the validation data to the controller:

```
• PIN Input                        Check or
• Validation ==> PINVERIF ==> •  No-Check
   Input                            Indication
```

PINVERIF determines if the data and the PIN are *algorithmically related* and notifies the program. This enables the program to determine with a high degree of reliability that the entered PIN is valid.

The algorithmic relationship is a complex one, involving both the data that serves as input to the algorithm, and several program-provided values that control the algorithm.

The 4700 controller and the 3624 Consumer Transaction Facility use the same algorithm. However, the PIN does not have to be in 3624 format; PINVERIF accepts PINs in other formats, as described below.

The 4700 Host Support program (Licensed Program 5668-989) contains a routine, BDKDPRS, that you can use to generate PINs and data that have the correct algorithmic relationship. BDKDPRS is described in detail in Chapter 8, "Host Support Encryption Routines."

Chapter 8, "Host Support Encryption Routines" also describes the PIN validation algorithm. You can use this description if you do not have access to the Host Support program and want to write your own PIN-generating program.

The program that is validating the customer-entered PIN must have access to the data and the controlling values. The program cannot derive the data and values from the PIN alone; generally, the program obtains them from the customer's identification card or from the operator handling the transaction. At a minimum, the customer or terminal operator would provide some form of identifier (in addition to the PIN) that the program can use to find the data and value that you have installed. Or, at the other extreme, the program could read *all* the required data and values from the terminal -- from a magnetically-encoded card, for example, or from the terminal's keyboard. Typically, however, a procedure part way between these two extremes is used: the customer's card contains an identifier, the validation data, and optionally one of the control values; all other control values are in a user-defined table indexed via the identifier. The examples later in this chapter illustrate this procedure.

Obtaining the PIN

A program can acquire a PIN from any of these sources:

- A 4704 display station with a nonencrypting PIN keypad accessory

- A 4704 display station with an encrypting PIN keypad accessory

- A 3624 consumer transaction facility.

In all cases, the program obtains the PIN as the result of issuing an input instruction. At the completion of the input operation, the PIN is in the program's segment storage. The delimiters surrounding the PIN differ depending on the input device and on what you specified during the controller configuration procedure. See the publications listed below for more information.

Note: In addition to the above, other PIN formats are supported.

PIN Source	Instruction	Described In
Nonencrypting PIN Keypad	LREAD KB	*4700 Loop and DCA Device Programming*
Encrypting PIN Keypad	LREAD KB	Next chapter, and *Loop and DCA Device Programming*
3624 Facility	LREAD CT	*3624 Programmer's Reference*

Note: The 3624 PIN is not delimited; it occupies the sixth through the thirteenth bytes of the 3624's transaction request message.

PIN Formats

Except for the nonencrypting PIN format, the customer-entered PIN is always combined with other information (such as a pad character) and enciphered under KP1 -- the PIN input protection key. This is represented as

eKP1(PIN)

where PIN is the entire *formatted* PIN, as described below. KP1 is generally the key stored in the encrypting terminal. The program requires access to this key enciphered under the third variant of the master key -- eKM3(KP1).

The diagrams below show how the PIN would look if it were not enciphered.

```
    |<——1 to 16 bytes in plaintext——>|
 ________________________________________
 |7F|                                    |7F| Segment storage
 ________________________________________
    |PIN...                             | Formatted PIN
    __________________________________
    | F0 F1 F2 F3 F4 F5 | Example (Customer PIN: 0 1 2 3 4 5)
    ___________________
_____________________________________________________________
PIN:     From one to sixteen EBCDIC characters.
```

Nonencrypting PIN Keypad Format

```
    |<——— 8 bytes    ————> |
 _________________________
 |7E|                     |7F| Segment storage
 _________________________
    |LEN|PIN...    ...PAD|SEQ| Formatted PIN
    ______________________
    |A0 12 34 56 78 9F FF 00 | Example
    ______________________       (Customer PIN: 0 1 2 3 4 5 6 7 8 9)
    |<—— enciphered ————>|
         under KP1
_____________________________________________________________
LEN:     Number of PIN characters entered; a 4-bit value from X'1'
         to X'D'.

PIN...:  From one to thirteen PIN characters; each is a 4-bit value
         from X'0' to X'9'.

...PAD:  From zero to twelve pad characters (thirteen minus the number
         of PIN characters equals the number of pad characters);
         each is a 4-bit value, always X'F'.

SEQ:     A one-byte sequence number, from X'00' to X'FF'.
```

Encrypting PIN Keypad Format

```
 |<——   8 bytes    ——>|
 ______________________
 |                    | Segment storage
 ______________________
 |PIN...        ...PAD| Formatted PIN
 ______________________
 |01 23 45 6E EE EE EE EE| Example (Customer PIN: 0 1 2 3 4 5 6)
 ______________________
 |<—— enciphered ———>|
       under KP1
________________________________________________________________
PIN...: From one to sixteen PIN characters; each is a four-bit value
        from X'0' to X'9'.

...PAD: From zero to fifteen pad characters (sixteen minus the number
        of PIN characters equals the number of pad characters); each
        is a four-bit value from X'0' to X'F'; all must be the same.
```

3624 PIN Format

```
 |<——   8 bytes    ——>|
 ______________________
 |                    | Segment storage
 ______________________
 |0|LEN|PIN...  ...PAD| Plaintext PIN
 ______________________
 |0000|PAN...         | Primary Account Number (PAN)
 ______________________
 |XOR... (PIN XOR PAN) | Formatted PIN
 ______________________
 |<—— enciphered ———>|
       under KP1
                        Example —
 ______________________
 |06 12 34 56 FF FF FF FF|  (Customer PIN: 1 2 3 4 5 6)
 ______________________
 |00 00 22 23 33 44 45 55|  (Customer PAN: 111 222 333 444 555)
 ______________________
 |06 12 16 75 CC BB BA AA|  Formatted PIN (PIN XOR PAN)
 ______________________
________________________________________________________________
0:      A 4-bit control field; always X'0'.

LEN:    Number of PIN characters entered; a 4-bit value from X'4'
        to X'C'.

PIN...: From four to twelve PIN characters; each is a 4-bit value
        from X'0' to X'9'.

...PAD: From two to ten pad characters (fourteen minus the number
        of PIN characters equals the number of pad characters); each
        4-bit character must be set to X'F'.

0000:   A 2-byte field; always X'0000'.

PAN...: Twelve 4-bit digits representing the "rightmost" (least
        significant) twelve digits of the primary account number.

XOR...: An exclusive-OR of the plaintext PIN and the PAN yields the
        formatted PIN.
```

ANSI PIN Format

```
|<——    8 bytes    ——>|
_______________________
|                     | Segment storage
_______________________
|SEQ|PIN...     ...PAD| Formatted PIN
_______________________
|00 00 12 34 56 EE EE EE| Example (Customer PIN: 1 2 3 4 5 6)
_______________________
|<—— enciphered ——>|
     under KP1
```

SEQ: A 2–byte sequence number assigned by the 3621 (or originating terminal 'or node) to each PIN prior to transmitting it to the controller. (The 3621 assigns the sequence number in ascending order, starting with zero. A program can use a DEVPARM instruction to change the sequence number.)

PIN...: From one to twelve PIN characters; each is a 4–bit value from X'0' to X'9'.

...PAD: From zero to eleven pad characters (twelve minus the number of PIN characters equals the number of pad characters); each is a four–bit value from X'0' to X'F'; all must be the same.

3621 PIN Format

PIN Data

The data accompanying the PIN can be obtained from the 4704 keyboard or from the magnetic stripe of a credit or other plastic card. The 4704's magnetic stripe unit reads cards prerecorded on Track 2 at 75 bpi, in accordance with American National Standards Institute (ANSI) Standard X4.16-1976.

The 3624 reads cards prerecorded on Tracks 2 and 3 as described in *3624 Programmer's Reference* (PR050). The 3624 can perform its own PIN validation at the time the customer enters the card and PIN.

Figure 6-1 illustrates the principal items of information that the program supplies when validating a PIN. (The following numbered items correspond to those in Figure 6-1.)

1. The *validation data* is the data that the controller compares (in a much-modified form) with the PIN itself during the validation operation. The validation data can be the customer's account number or any other identifying number chosen by the institution. You can run the BDKDPRS routine to generate a PIN that is correctly related to the validation data. The program pads the validation data on the right, if necessary, to make the validation data eight bytes long.

2. If an arbitrarily-specified number is to be read as a customer-assigned PIN, an additional value must be recorded on the customer's card. This *offset data* reflects the difference between the PIN that is generated from the validation data, and the PIN actually assigned to the customer. You can use the BDKDPRS routine to generate the proper offset data.

3. The *validation key* is the key you select for input to BDKDPRS. You provide this key (KPv) to BDKDPRS in plaintext, but the program provides the key enciphered under the third variant of the master key. It is your responsibility to create eKm3(KPv) and make it available to the validating program.

4. A *decimalization table* is a string of sixteen packed decimal digits that the controller uses to modify the enciphered validation data. You supply this arbitrarily-selected table to BDKDPRS, and to the validating program.

5. The *check length* indicates how many digits of the customer-entered PIN are to be processed (compared) by PINVERIF. The minimum is one, and the maximum is sixteen. The check length also governs how many digits of offset data (if present) are to be processed by PINVERIF.

6. The program passes PINVERIF the *PIN* in the same format as it was received -- that is, enciphered under the input PIN protection key, KP1 (or in plaintext if the PIN is obtained from a nonencrypting PIN keypad).

7. When the PIN is in a 3624 format, the *pad character* specifies the digit that the 3624 uses to pad the PIN.

8. The enciphered *protection key* is the key KP1 enciphered under the third variant of the controller's master key. You must create eKM3(KP1) and make it available to the validating program.

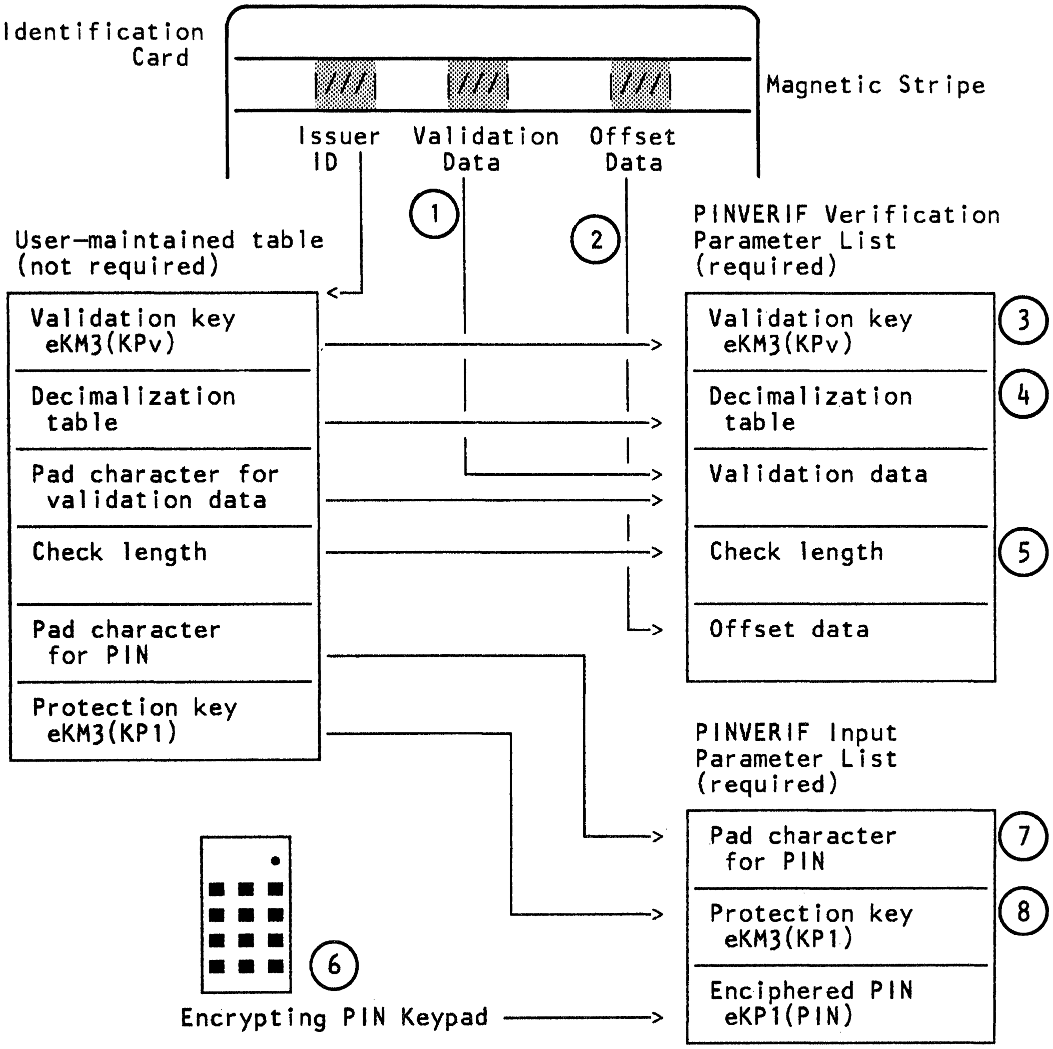


Figure 6-1. Information Required for PIN Validation

Example 6. Validating a PIN

The following example shows how a program can use the validation data together with the 4700 cryptographic facilities to validate a PIN.

In this example, the 4700 controller application program can validate enciphered PINs received from any of three encrypting terminals attached to it (Figure 6-2):

- 4704 display stations *X* and *Y*, each equipped with an encrypting PIN keypad and a magnetic stripe unit.

- 3624 consumer transaction facility *Z*.

Before program execution, the installation's security specialists have defined and installed the following:

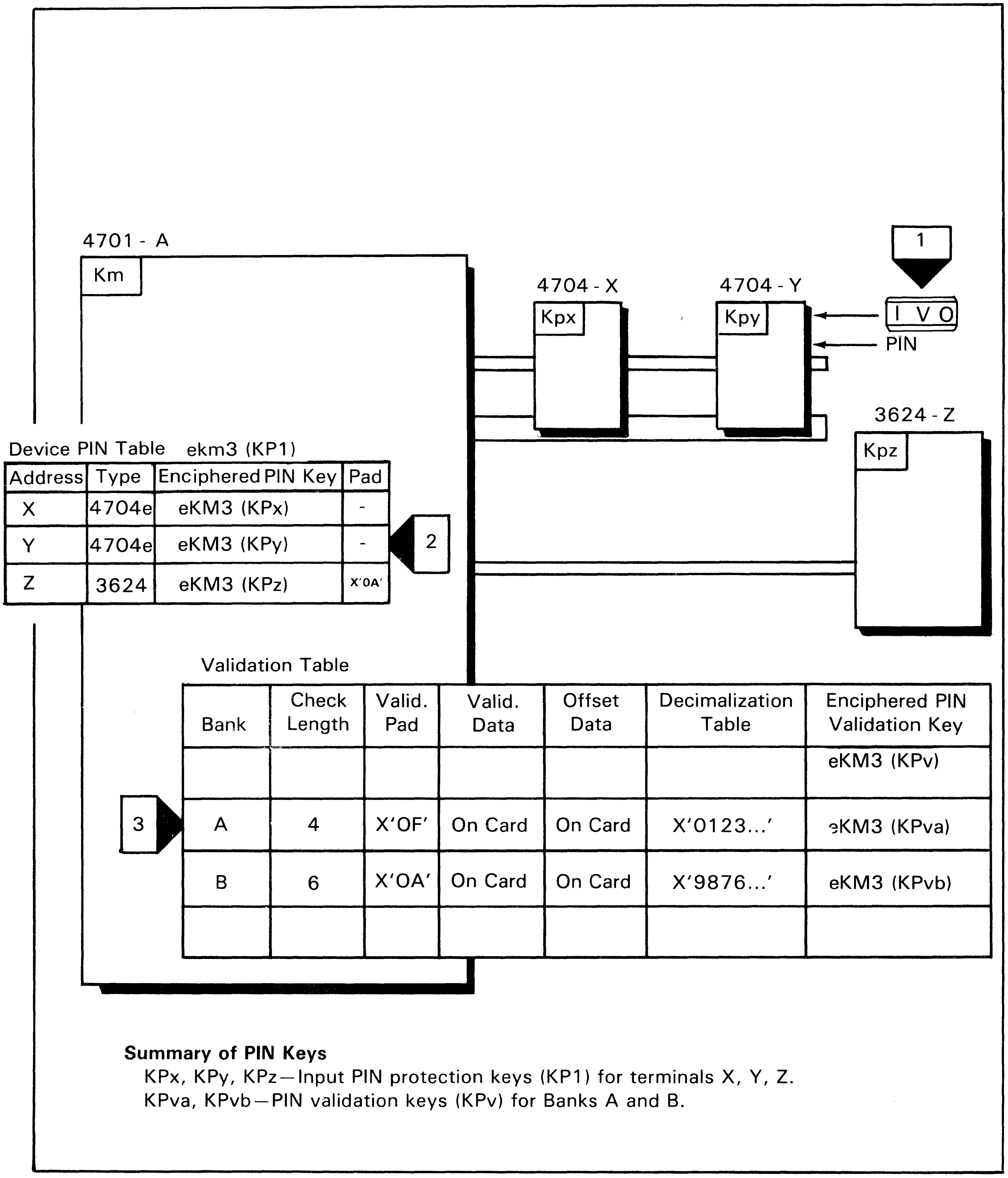
- KM, the controller's master key. KM is installed manually in the controller.

- KPx, KPy, and KPz, input PIN protection keys for the three encrypting terminals. KPx and KPy are installed manually in their respective encrypting keypads (see Chapter 7, "Using the Encrypting PIN Keypad"); KPz is transmitted to the 3624 in its load image.

- eKM3(KPx), eKM3(KPy), and eKM3(KPz), installed in *PIN Table*.

- KPva, a PIN validation key for PINs issued by the 4700's bank (A), and KPvb, a PIN validation key for PINs issued by a second bank, B, whose customers are permitted to use Bank A's facilities.

- eKM3(KPva) and eKM3(KPvb), installed in *Validation Table*.



Device PIN Table ekm3 (KP1)

Address	Type	Enciphered PIN Key	Pad
X	4704e	eKM3 (KPx)	-
Y	4704e	eKM3 (KPy)	-
Z	3624	eKM3 (KPz)	X'0A'

Validation Table

Bank	Check Length	Valid. Pad	Valid. Data	Offset Data	Decimalization Table	Enciphered PIN Validation Key
						eKM3 (KPv)
A	4	X'OF'	On Card	On Card	X'0123...'	eKM3 (KPva)
B	6	X'OA'	On Card	On Card	X'9876...'	eKM3 (KPvb)

Summary of PIN Keys

KPx, KPy, KPz—Input PIN protection keys (KP1) for terminals X, Y, Z.
KPva, KPvb—PIN validation keys (KPv) for Banks A and B.

Figure 6-2. Validating a PIN

1. The program receives input from terminal Y's operator, requesting a banking transaction that requires PIN validation. The program:

   - Signals the operator to obtain the magnetic stripe data.

   - Enables the magnetic stripe unit and the encrypting keypad.

   - Reads the magnetic stripe data and the enciphered PIN.

2. The program accesses terminal Y's entry in the PIN Table and obtains the enciphered protection key. The program now has

   - eKPy(PIN)

   - eKM3(KPy)

   and can set up PINVERIF's input parameter list.

3. The program uses the first of the three fields of magnetic stripe data (the bank identifier) to find the proper entry in the Validation Table. In this example, the magnetic stripe data identifies the customer as a customer of Bank A. The program extracts the check length, decimalization table, and enciphered PIN validation key from the table and places them into PINVERIF's verification parameter list. The program then:

   - Determines that the validation data and the validation offset data are to be taken from the magnetic stripe data.

   - Pads the validation data with the table's pad character.

   - Places the offset data and the padded validation data into the parameter list.

The program can now issue the PINVERIF instruction and tell the operator if the PIN is acceptable.

PIN Translation

PIN translation is intended for use by programs that are exchanging PINs with remote locations.

Using the PINTRANS instruction, your program can reencipher the PIN and/or change the PIN from the format in which your program acquired the PIN to a format usable by the other program.

PINTRANS, like PINVERIF, processes enciphered PINs without revealing the PIN in plaintext. PINTRANS accepts PINs in any of the following formats:

- nonencrypting PIN keypad format
- encrypting PIN keypad format (enciphered under KP1)
- 3624 PIN format (enciphered under KP1)
- ANSI PIN format (enciphered under KP1)
- 3621 PIN format (enciphered under KP1)

and can produce the same PIN in any of the following enciphered formats:

- encrypting PIN keypad format
- 3624 PIN format
- ANSI PIN format
- 3621 PIN format

Before transmitting a PIN to another program, you can use PINTRANS to reencipher the PIN under a key usable to the receiving program. This key is referred to as the *output* PIN protection key, KP2, and must be available to the program enciphered under the first variant of the controller's master key.

PINTRANS operates in the following manner when the input PIN is in *nonencrypting* PIN keypad format. Note that the PIN is enciphered under an output PIN protection key (KP2):

```
• PIN
• eKM1(KP2) ==> PINTRANS ==> • eKP2(PIN)
```

PINTRANS operates in the following manner when the input PIN is in one of the *encrypting* PIN formats. Reenciphering can occur under either the same key (KP1) *or* under a new key (KP2). Note that PINs in encrypting PIN format can be translated without reenciphering under a new key. You could use this form of PINTRANS to change the format and/or pad character in a formatted PIN.

Reenciphering Under the Same Key:

```
• eKP1(PIN)
• eKM3(KP1) ==> PINTRANS ==> • eKP1(PIN)
```

Reenciphering Under a New Key:

```
• eKP1(PIN)
• eKM3(KP1) ==> PINTRANS ==> • eKP2(PIN)
• eKM1(KP2)
```

The enciphered PIN must conform to one of the PIN formats described above.

Example 7. Translating a PIN

In the previous example, the 4700 controller application program at Bank A could verify PINs issued by Banks A and B. In Example 7, customers of remote banks C and D are also permitted to conduct transactions through Bank A's facilities.

The program does not have access to the information necessary to validate these additional PINs. When a customer of Bank C or D enters a transaction, the program sends the PIN and its accompanying data to the customer's bank for validation there.

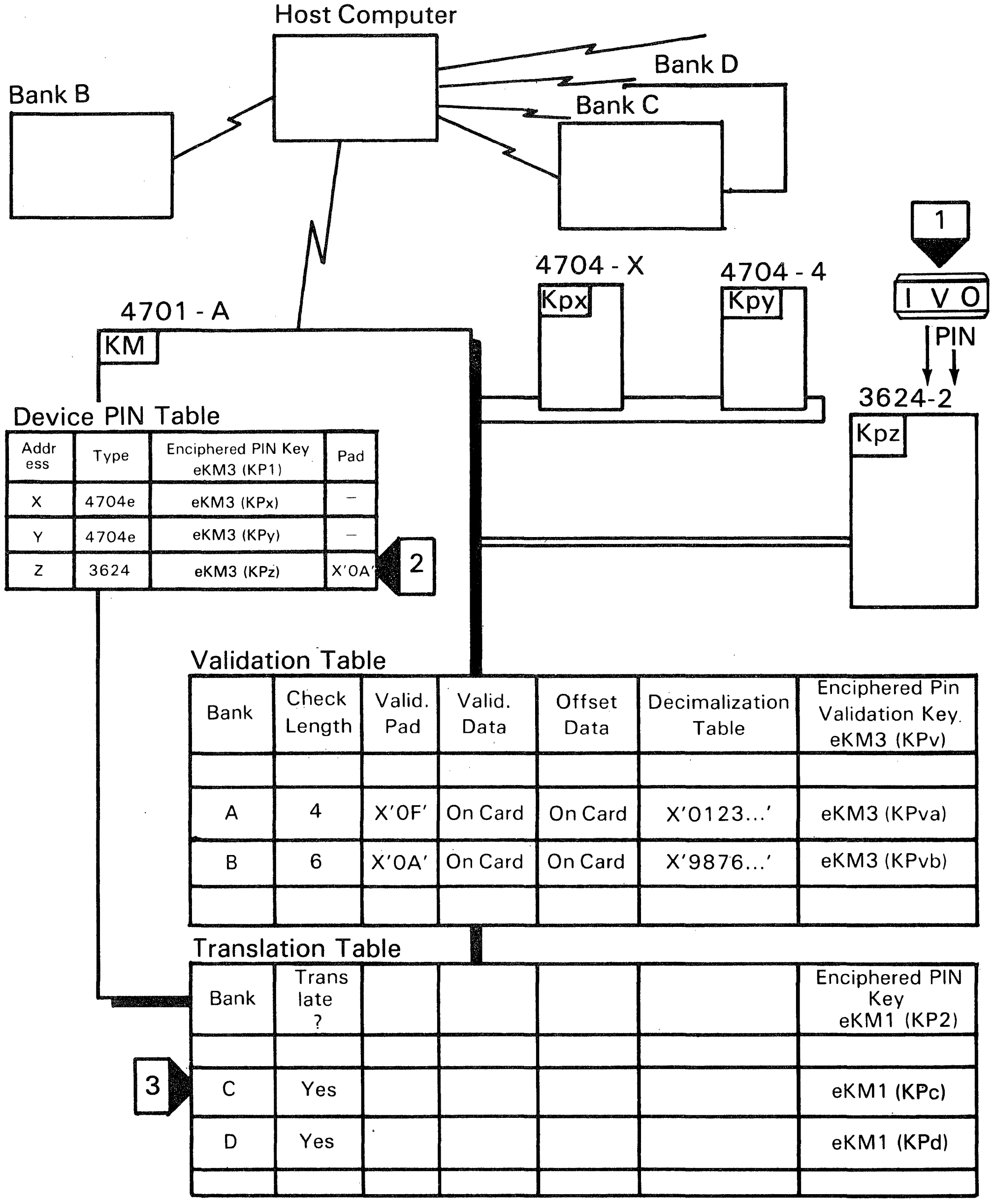
Bank A's 4700 controller is attached to a host computer that acts as an intermediary between Bank A and the remote banks. Although not illustrated in this example, the host computer could relay validation requests in either direction. If Bank A or B customers attempted a transaction at Banks C or D, the host computer would request the same type of PIN validation operation shown in the previous example. The 4700 physical configuration is the same as in the previous example: Two 4704 display stations (X and Y) and one 3624 consumer transaction facility (Z) are attached to A's controller.

In addition to the input PIN protection keys and the validation keys already defined (KPx, KPy, KPz, KPva, and KPvb), Bank A's security specialists have also defined two additional output PIN protection keys (KP2):

- KPc -- protection key for PINs sent to C
- KPd -- protection key for PINs sent to D

Bank A has enciphered these keys under the first variant of the 4701's master key, and made the results available to the program. These enciphered keys must, of course, be available to C and D.

Bank A has also defined the PIN formats required by C and D -- in this example, 3624 format for both banks. These inputs are shown in Figure 6-3 as a separate PIN translation table, but note that this information could easily be combined with the PIN validation table.



Device PIN Table

Addr ess	Type	Enciphered PIN Key eKM3 (KP1)	Pad
X	4704e	eKM3 (KPx)	–
Y	4704e	eKM3 (KPy)	–
Z	3624	eKM3 (KPz)	X'0A'

Validation Table

Bank	Check Length	Valid. Pad	Valid. Data	Offset Data	Decimalization Table	Enciphered Pin Validation Key eKM3 (KPv)
A	4	X'0F'	On Card	On Card	X'0123...'	eKM3 (KPva)
B	6	X'0A'	On Card	On Card	X'9876...'	eKM3 (KPvb)

Translation Table

Bank	Trans late ?					Enciphered PIN Key eKM1 (KP2)
C	Yes					eKM1 (KPc)
D	Yes					eKM1 (KPd)

Summary of PIN Keys

KPx, KPy, KPz - Input PIN protection keys (KP1) for X, Y, and Z
KPva, KPvb - PIN validation keys (KPv) for Banks A and B
KPc, KPd - Output PIN protection keys (KP2) for PINs sent to C and D

Figure 6-3. Translating a PIN

1. The program receives a transaction request message from the 3624 consumer transaction facility (terminal Z). The program obtains the customer's PIN (enciphered under KPz) and the accompanying magnetic stripe data.

2. The program finds terminal Z's entry in the device PIN table and obtains eKM3(KPz) and the pad character. The program now has all required input for the INPPAR parameter list (see the PINVERIF instruction for an explanation of the INPPAR field names):

   - INPIN = eKPz(PIN) -- the enciphered PIN
   - INPINKEY = eKM3(KPz) -- the enciphered PIN protection key
   - INPINPAD = X'0A' -- the pad character A
   - INPINTYP = X'40' -- the input PIN format

3. The program uses the identifier portion of the magnetic stripe data (which identifies Bank C) to find the proper entry in the translation table. This entry tells the program that an online PIN validation is required (that is, the program cannot itself validate the PIN). This entry also provides eKM1(KPc).

4. The program sets up the PINTRANS translation parameter list as follows, and issues PINTRANS:

   - TNPINFLG = X'40' -- reencipher under new key
   - TNPINPAD = X'0A' -- use pad character X'A'
   - TNPINKEY = eKM1(KPc) -- reencipher under KPc

5. The program combines the reenciphered formatted PIN with the magnetic stripe data and prepares to transmit the message to the host computer. This preparation includes generating a session key, reenciphering from the master key to the host's cross-domain key, and establishing a session with the host program -- as shown previously in Example 4.

6. The host computer relays the transaction to Bank C, where a program uses the message to validate the PIN. Bank C returns a response through the host computer back to A. The program at Bank A can now tell the 3624 whether to perform the transaction.

Managing PIN Keys

As you have seen from the descriptions of PINTRANS, PINVERIF, and the previous examples, you maintain PIN keys in three separate classes:

- input PIN protection keys (KP1)
- output PIN protection keys (KP2)
- PIN validation keys (KPv)

Input PIN protection keys protect PINs that the program receives. Examples include the keys that are transmitted to a 3624 or that are manually loaded into a 4704 encrypting PIN keypad.

Output PIN protection keys are used to protect formatted PINs so that they can be securely transmitted to programs at remote locations, where the PIN can be validated or matched with a data base of enciphered PINs.

The PIN validation algorithm requires a PIN *validation key* in order to generate a PIN check number, as described in Chapter 8, "Host Support Encryption Routines." The same key that is used for validation is also used when you initially generate the customer PINs.

The number of PIN protection keys and PIN validation keys needed depends on the type of PIN operations being performed and the number of institutions and locations involved in the PIN operations.

If no PIN handling operations are performed at a particular controller, no PIN keys are required.

PIN protection keys are typically defined on the basis of PIN sources and PIN destinations. An application might maintain a PIN protection key for each input device or program from which it receives PINs, and a PIN protection key for each program to which it transmits PINs.

For example, suppose a controller is supporting several different PIN input devices or terminals: 3624s, encrypting PIN keypads, and nonencrypting PIN keypads. Each input device, with the exception of the nonencrypting PIN keypad, is loaded with a different PIN protection key. All PINs are to be transmitted to a host application program where they are validated by comparing them against a data base of PINs enciphered under a single PIN key. In this case, the application program in the controller would define an input PIN protection key for each attached 3624 and encrypting PIN keypad, and a single output PIN protection key for the host application program.

When the controller receives the PIN, it would reformat and translate the PIN from encryption under one of the input PIN protection keys to encryption under the output PIN protection key. If, in this same example, each input device were loaded with the same key, the application would need to support only a single input PIN protection key and a single output protection key.

It is not necessary for an application program to support output PIN protection keys unless it changes the key under which a PIN is enciphered before forwarding it to another program. That is, if all PINs are originally enciphered in the same key and the program merely translates them into a common format without changing the key, only a single input PIN protection key is needed. The

PINTRANS instruction enables you to specify that the same key be used to encipher the output PIN as was used to protect the input PIN.

PIN validation keys are typically defined by each institution. An institution may use a single PIN validation key to verify all of it's customers' PINs, or multiple PIN validation keys to support unique subsets of customers. A program that performs PIN validation for several institutions would maintain a PIN validation key or set of PIN validation keys for each institution it supports. When the program validates the PIN, the program selects the appropriate PIN validation key on the basis of identification card data or account information.

Key Storage

You maintain PIN keys in enciphered form in storage or on a diskette. In order to use them in the PINTRANS and PINVERIF instructions, you must store the PIN keys under the appropriate variant of the controller master key. You store all input protection keys and PIN validation keys under the third variant of the master key, and all output PIN protection keys under variant one. If a particular PIN protection key serves as both input and output (and never as output alone), you need not maintain two copies of the key enciphered under both variant one and variant three; the PINTRANS instruction allows the same key to be used as an input and an output key.

Key Generation and Distribution

PIN keys are static keys generated by your key generation system. They are initially generated as plaintext keys and then enciphered by the key generation system, using the appropriate variant of the destination program's master key. You can encipher a PIN key using either the ENCODE or ENCIPHER instruction, depending on how controller master keys are stored in the key generation system.

Distribute PIN keys only after you have enciphered them under the appropriate variant of the controller master key. Enciphered PIN keys may be stored on a diskette, transmitted (downloaded) from a host system, or manually entered through a program-provided key entry procedure.

Protecting PINs

One of the obvious objectives of encryption is to protect a customer's PIN from unauthorized disclosure. The following section points out some of the problems that you may face in trying to protect PINs, and suggests some techniques you might use to overcome these problems. Remember that the problems described below are samples, and may not be the only ones you will face.

The easiest way for an adversary to obtain a PIN is to intercept the PIN in plaintext. It is therefore important to limit the places in the system where PINs appear in plaintext.

A PIN entered at an encrypting PIN keypad or at a 3624 always enters the controller in enciphered form. With the PINVERIF instruction (and its equivalent function in the 3624), the PIN can be validated without requiring transmission or storage of the PIN elsewhere in the system. In instances where the PIN must be validated at a remote location, PINTRANS should be used to put the PIN in its final enciphered form before leaving the controller. The PIN need not be deciphered at the remote location; the enciphered PIN can be matched against enciphered PINs that reside on the remote location's data base.

Dictionary Attacks

Even when using all these techniques, you should take further precautions against *dictionary attacks.*

If an adversary knows that his PIN always appears the same when enciphered, he may watch for other customer accounts whose enciphered PINS match his. (Such duplicates are not uncommon when institutions assign short PINs.) When he has identified one, he knows that that customer's plaintext PIN may be the same as his. This attack can be expanded to build a paired dictionary of plaintext PINs and enciphered PINs.

Protection against dictionary attacks generally takes the form of including variable data with the PIN before enciphering it. This increases the number of enciphered PIN synonyms per plaintext PIN, proportional to the variability of the data. This in turn makes the management of the attack dictionary extremely difficult.

The encrypting PIN keypad automatically includes a one-byte binary sequence counter with the PIN before enciphering it. This creates 256 enciphered PIN synonyms for each clear PIN. Note that although this discourages dictionary attacks, this enciphered PIN format is of little use if enciphered PINS are to be compared at the bank's data base. All 256 synonyms would need to be stored for each customer PIN. For host validation of PINS entered this way, use the PINTRANS instruction to translate all such PINs into 3624 format.

The 3624 PIN format does not include variable data with the PIN. It is useful for enciphered PIN comparisons at the data base; however, it in itself is not secure against dictionary attacks. To provide dictionary attack protection during transmission, include an enciphered PIN in 3624 format within a message that you then encipher in cipher block chaining mode (ENCIPHER). Note that this is effective only if the initial chaining value and/or preceding data in the message data field are varied with each transmission.

Exhaustion Attacks

An *exhaustion attack* is the process of determining a customer PIN through trial and error. These attacks range from a simple attack at the consumer interface (trying different PINS with a stolen card) to combined dictionary and exhaustion attacks. They all have one requirement in common: the need to enter a transaction with plaintext PIN input.

Simple attacks can be defeated rather easily: have the program limit the number of PIN transaction entries. More complex attacks include using the controller offline to build a dictionary of enciphered and plaintext PINs, and using the controller offline to validate PINs. To discourage such attacks, you can implement a program that detects excessive executions of the PINVERIF and PINTRANS instructions. If PINs are validated in the controller, this program could audit counts of successful and unsuccessful PIN validations and PIN translations. You could then check these counts to detect abnormally high proportions of unsuccessful attempts, or your program could maintain a threshold value of unsuccessful attempts. The controller maintains three *statistical counters* to aid you in making such audits:

- *Counter 9-11* increments each time a PINVERIF instruction completes successfully, up to the point where the PIN is actually validated (SMSCCD = X'01', or SMSCCD = X'02' and SMSDST = X'2010'). This is the total number of PIN validations attempted.

- *Counter 12-14* increments each time a PINVERIF instruction completes successfully, but results in a "PIN not valid" status (SMSCCD = X'02' and SMSDST = X'2010'). This is the number of unsuccessful PIN validations.

- *Counter 15-17* increments each time a PINTRANS instruction completes successfully -- that is, completes with a condition code of 1 (SMSCCD = X'01').

Statistical counters are described further in Appendix F, "Statistical Counters."

You could augment this procedure by keeping track of recently used account numbers. Excessive use of the same account number for unsuccessful PIN validations could also be monitored by the program.

If PINs are being checked at the host computer, the program could simply monitor the number of times that PINTRANS is executed. Auditing this count with respect to the number of transactions received at the host could disclose unauthorized use of the controller.

If responses are paired with PIN translations, the program could disable the PINTRANS function when valid host responses were not present. This active protection could be used even when PINs are validated in the controller, provided that all transactions are online.

The expense of developing such a program would likely be small, compared to the value of the protection it provides.

Chapter 7. Using the Encrypting PIN Keypad

The encrypting PIN keypad (Figure 7-1, left) is a 4704 display station accessory. It enciphers a PIN in accordance with the United States National Bureau of Standards data encryption algorithm, and sends the enciphered PIN to the 4704 control module.

The encrypting PIN keypad has keys for numbers zero through nine, and two special keys labeled ERASE and END. The ERASE key clears the entered PIN, and the END key delimits the end of the entered PIN.

The numeric keys are labeled with alphabetic characters. The keypad can be ordered with either of two labeling schemes; either with the label QZ next to the 1 key (as shown in Figure 7-1, right) or with the label QZ next to the 0 key.

The QZ-with-1 format conforms to proposed American National Standards Institute (ANSI) Standard X4.A11. However, if your installation already includes PIN keypads having the QZ-with-0 format, it is important that you maintain compatibility with that format.

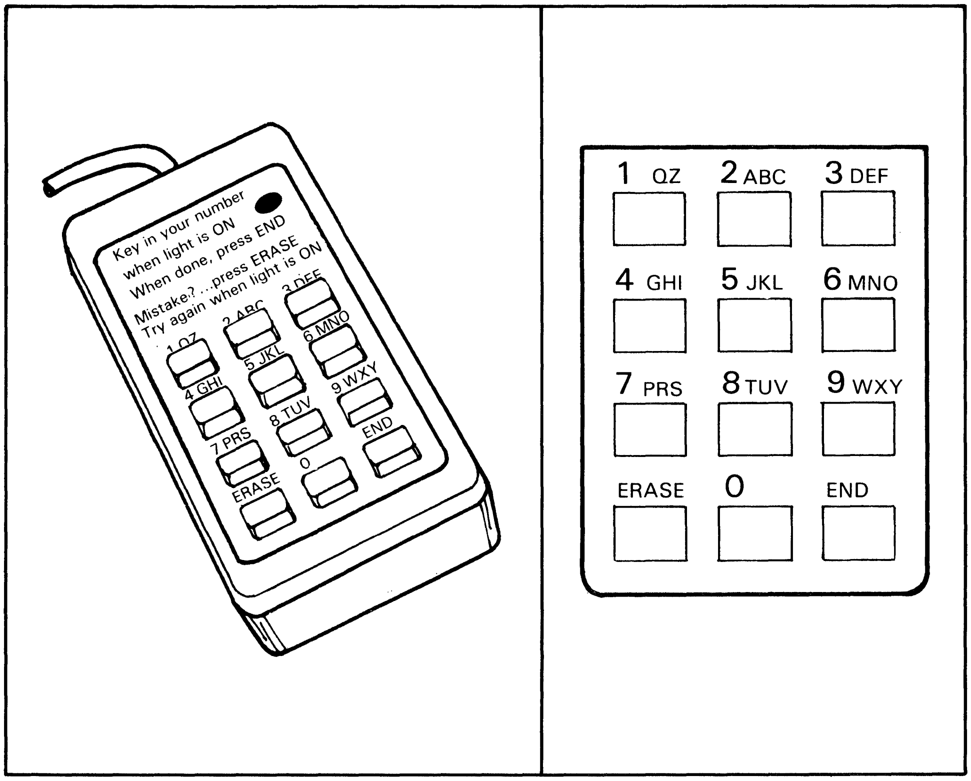


Figure 7-1. Encrypting PIN Keypad

Entering the Key

You load the PIN keypad key (KP1) through the keypad itself, although the attached 4704 is also required. The entire process involves the following steps:

- Ensure that the 4704 and the keypad have been set up and checked out in accordance with *4704 Setup Instructions.*

- Apply power to the 4704 (see *4704 Operating Instructions* if you don't know how to operate the 4704).

- Activate the keypad switch. The keypad switch is on the right side of the keypad (Figure 7-2). Insert a small tool (such as a paper clip or small screwdriver) into the switch opening and slide the switch to the *right* -- that is, toward the cable end of the keypad.

- Press the 4704 ALT and TEST keys simultaneously.

- Press the 4704 ALT and PIN Keypad Test keys simultaneously. The keypad indicator should light.

- Convert KP1 to a 24-key format, as described below. Remember to start with KP1 -- not with eKM3(KP1). You install the latter in the controller for the benefit of using programs, but you must load KP1 into the keypad in *unenciphered* form.

- Load KP1. The indicator light should go out after you enter the 24th keystroke. The keypad automatically replaces the old key after you enter the 24th keystroke.

- Return the switch on the side of the keypad to its original position.

  Note: To *erase* the key, enable the keypad, activate the keypad switch, and press the END key.

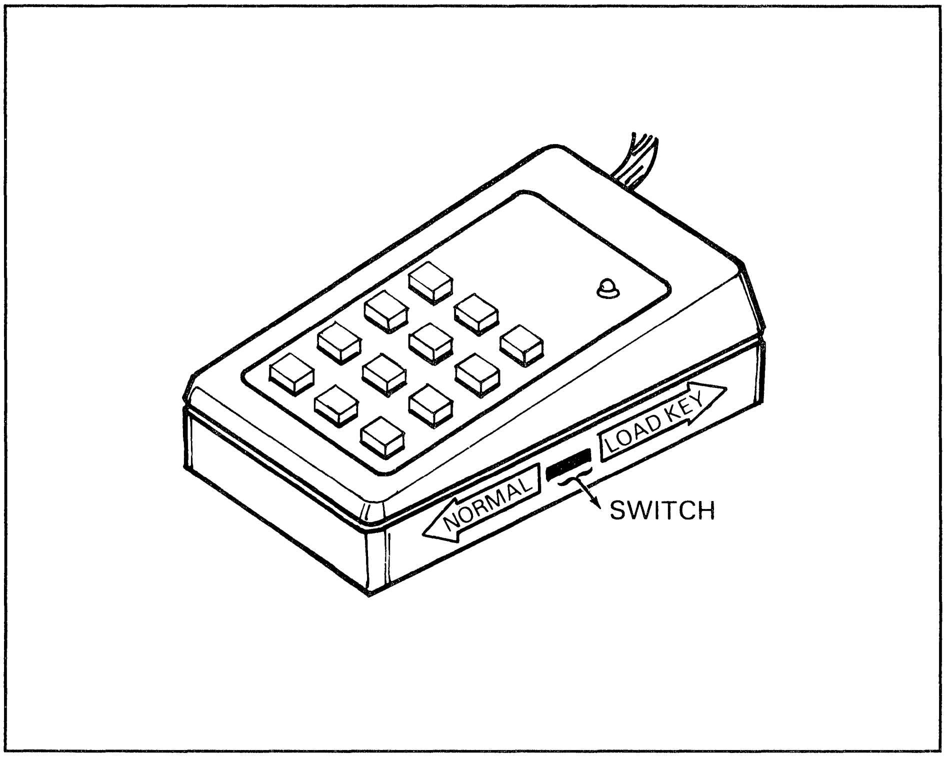


Figure 7-2. Encrypting PIN Keypad Switch

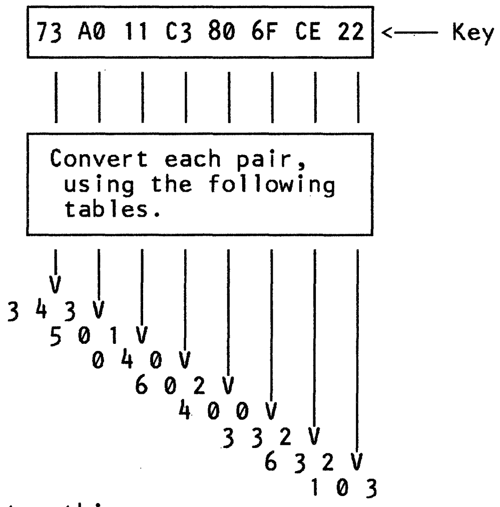
The indicator blinks if you make a mistake entering the key. Press the ERASE key and try again. If the keypad still blinks, the problem may be caused by:

- Incorrect conversion from hexadecimal format
- Defective keypad (see *4704 Operating Instructions)*
- Wrong key (bad parity)

Converting KP1 to 24-Key Format

As described earlier, you initially generate all keys as eight-byte values. In the case of input PIN protection keys (KP1) for encrypting PIN keypads, you must also convert that eight-byte value into 24 keystroke values. To express the key in the form of keystrokes, write the key in hexadecimal, then use the tables in Figure 7-3 to convert each pair of hexadecimal digits to three keystroke values.

For example:



```
 73 A0 11 C3 80 6F CE 22 <—— Key

  Convert each pair,
   using the following
   tables.

  V
3 4 3 V
   5 0 1 V
      0 4 0 V
         6 0 2 V
            4 0 0 V
               3 3 2 V
                  6 3 2 V
                     1 0 3
Enter this sequence
 on keypad ==> 3 4 3 5 0 1 0 4 0 6 0 2 4 0 0 3 3 2 6 3 2 1 0 3
```

X'00'	==>	0 0 1	X'20'	==>	1 0 0	X'40'	==>	2 0 0	X'60'	==>	3 0 1
X'01'	==>	0 0 1	X'21'	==>	1 0 0	X'41'	==>	2 0 0	X'61'	==>	3 0 1
X'02'	==>	0 0 2	X'22'	==>	1 0 3	X'42'	==>	2 0 3	X'62'	==>	3 0 2
X'03'	==>	0 0 2	X'23'	==>	1 0 3	X'43'	==>	2 0 3	X'63'	==>	3 0 2
X'04'	==>	0 1 0	X'24'	==>	1 1 1	X'44'	==>	2 1 1	X'64'	==>	3 1 0
X'05'	==>	0 1 0	X'25'	==>	1 1 1	X'45'	==>	2 1 1	X'65'	==>	3 1 0
X'06'	==>	0 1 3	X'26'	==>	1 1 2	X'46'	==>	2 1 2	X'66'	==>	3 1 3
X'07'	==>	0 1 3	X'27'	==>	1 1 2	X'47'	==>	2 1 2	X'67'	==>	3 1 3
X'08'	==>	0 2 0	X'28'	==>	1 2 1	X'48'	==>	2 2 1	X'68'	==>	3 2 0
X'09'	==>	0 2 0	X'29'	==>	1 2 1	X'49'	==>	2 2 1	X'69'	==>	3 2 0
X'0A'	==>	0 2 3	X'2A'	==>	1 2 2	X'4A'	==>	2 2 2	X'6A'	==>	3 2 3
X'0B'	==>	0 2 3	X'2B'	==>	1 2 2	X'4B'	==>	2 2 2	X'6B'	==>	3 2 3
X'0C'	==>	0 3 1	X'2C'	==>	1 3 0	X'4C'	==>	2 3 0	X'6C'	==>	3 3 1
X'0D'	==>	0 3 1	X'2D'	==>	1 3 0	X'4D'	==>	2 3 0	X'6D'	==>	3 3 1
X'0E'	==>	0 3 2	X'2E'	==>	1 3 3	X'4E'	==>	2 3 3	X'6E'	==>	3 3 2
X'0F'	==>	0 3 2	X'2F'	==>	1 3 3	X'4F'	==>	2 3 3	X'6F'	==>	3 3 2
X'10'	==>	0 4 0	X'30'	==>	1 4 1	X'50'	==>	2 4 1	X'70'	==>	3 4 0
X'11'	==>	0 4 0	X'31'	==>	1 4 1	X'51'	==>	2 4 1	X'71'	==>	3 4 0
X'12'	==>	0 4 3	X'32'	==>	1 4 2	X'52'	==>	2 4 2	X'72'	==>	3 4 3
X'13'	==>	0 4 3	X'33'	==>	1 4 2	X'53'	==>	2 4 2	X'73'	==>	3 4 3
X'14'	==>	0 5 1	X'34'	==>	1 5 0	X'54'	==>	2 5 0	X'74'	==>	3 5 1
X'15'	==>	0 5 1	X'35'	==>	1 5 0	X'55'	==>	2 5 0	X'75'	==>	3 5 1
X'16'	==>	0 5 2	X'36'	==>	1 5 3	X'56'	==>	2 5 3	X'76'	==>	3 5 2
X'17'	==>	0 5 2	X'37'	==>	1 5 3	X'57'	==>	2 5 3	X'77'	==>	3 5 2
X'18'	==>	0 6 1	X'38'	==>	1 6 0	X'58'	==>	2 6 0	X'78'	==>	3 6 1
X'19'	==>	0 6 1	X'39'	==>	1 6 0	X'59'	==>	2 6 0	X'79'	==>	3 6 1
X'1A'	==>	0 6 2	X'3A'	==>	1 6 3	X'5A'	==>	2 6 3	X'7A'	==>	3 6 2
X'1B'	==>	0 6 2	X'3B'	==>	1 6 3	X'5B'	==>	2 6 3	X'7B'	==>	3 6 2
X'1C'	==>	0 7 0	X'3C'	==>	1 7 1	X'5C'	==>	2 7 1	X'7C'	==>	3 7 0
X'1D'	==>	0 7 0	X'3D'	==>	1 7 1	X'5D'	==>	2 7 1	X'7D'	==>	3 7 0
X'1E'	==>	0 7 3	X'3E'	==>	1 7 2	X'5E'	==>	2 7 2	X'7E'	==>	3 7 3
X'1F'	==>	0 7 3	X'3F'	==>	1 7 2	X'5F'	==>	2 7 2	X'7F'	==>	3 7 3

Figure 7-3 (Part 1 of 2). Converting KP1 from Hexadecimal to Keystroke Format

X'80'	==>	4 0 0	X'A0'	==>	5 0 1	X'C0'	==>	6 0 1	X'E0'	==>	7 0 0
X'81'	==>	4 0 0	X'A1'	==>	5 0 1	X'C1'	==>	6 0 1	X'E1'	==>	7 0 0
X'82'	==>	4 0 3	X'A2'	==>	5 0 2	X'C2'	==>	6 0 2	X'E2'	==>	7 0 3
X'83'	==>	4 0 3	X'A3'	==>	5 0 2	X'C3'	==>	6 0 2	X'E3'	==>	7 0 3
X'84'	==>	4 1 1	X'A4'	==>	5 1 0	X'C4'	==>	6 1 0	X'E4'	==>	7 1 1
X'85'	==>	4 1 1	X'A5'	==>	5 1 0	X'C5'	==>	6 1 0	X'E5'	==>	7 1 1
X'86'	==>	4 1 2	X'A6'	==>	5 1 3	X'C6'	==>	6 1 3	X'E6'	==>	7 1 2
X'87'	==>	4 1 2	X'A7'	==>	5 1 3	X'C7'	==>	6 1 3	X'E7'	==>	7 1 2
X'88'	==>	4 2 1	X'A8'	==>	5 2 0	X'C8'	==>	6 2 0	X'E8'	==>	7 2 1
X'89'	==>	4 2 1	X'A9'	==>	5 2 0	X'C9'	==>	6 2 0	X'E9'	==>	7 2 1
X'8A'	==>	4 2 2	X'AA'	==>	5 2 3	X'CA'	==>	6 2 3	X'EA'	==>	7 2 2
X'8B'	==>	4 2 2	X'AB'	==>	5 2 3	X'CB'	==>	6 2 3	X'EB'	==>	7 2 2
X'8C'	==>	4 3 0	X'AC'	==>	5 3 1	X'CC'	==>	6 3 1	X'EC'	==>	7 3 0
X'8D'	==>	4 3 0	X'AD'	==>	5 3 1	X'CD'	==>	6 3 1	X'ED'	==>	7 3 0
X'8E'	==>	4 3 3	X'AE'	==>	5 3 2	X'CE'	==>	6 3 2	X'EE'	==>	7 3 3
X'8F'	==>	4 3 3	X'AF'	==>	5 3 2	X'CF'	==>	6 3 2	X'EF'	==>	7 3 3
X'90'	==>	4 4 1	X'B0'	==>	5 4 0	X'D0'	==>	6 4 0	X'F0'	==>	7 4 1
X'91'	==>	4 4 1	X'B1'	==>	5 4 0	X'D1'	==>	6 4 0	X'F1'	==>	7 4 1
X'92'	==>	4 4 2	X'B2'	==>	5 4 3	X'D2'	==>	6 4 3	X'F2'	==>	7 4 2
X'93'	==>	4 4 2	X'B3'	==>	5 4 3	X'D3'	==>	6 4 3	X'F3'	==>	7 4 2
X'94'	==>	4 5 0	X'B4'	==>	5 5 1	X'D4'	==>	6 5 1	X'F4'	==>	7 5 0
X'95'	==>	4 5 0	X'B5'	==>	5 5 1	X'D5'	==>	6 5 1	X'F5'	==>	7 5 0
X'96'	==>	4 5 3	X'B6'	==>	5 5 2	X'D6'	==>	6 5 2	X'F6'	==>	7 5 3
X'97'	==>	4 5 3	X'B7'	==>	5 5 2	X'D7'	==>	6 5 2	X'F7'	==>	7 5 3
X'98'	==>	4 6 0	X'B8'	==>	5 6 1	X'D8'	==>	6 6 1	X'F8'	==>	7 6 0
X'99'	==>	4 6 0	X'B9'	==>	5 6 1	X'D9'	==>	6 6 1	X'F9'	==>	7 6 0
X'9A'	==>	4 6 3	X'BA'	==>	5 6 2	X'DA'	==>	6 6 2	X'FA'	==>	7 6 3
X'9B'	==>	4 6 3	X'BB'	==>	5 6 2	X'DB'	==>	6 6 2	X'FB'	==>	7 6 3
X'9C'	==>	4 7 1	X'BC'	==>	5 7 0	X'DC'	==>	6 7 0	X'FC'	==>	7 7 1
X'9D'	==>	4 7 1	X'BD'	==>	5 7 0	X'DD'	==>	6 7 0	X'FD'	==>	7 7 1
X'9E'	==>	4 7 2	X'BE'	==>	5 7 3	X'DE'	==>	6 7 3	X'FE'	==>	7 7 2
X'9F'	==>	4 7 2	X'BF'	==>	5 7 3	X'DF'	==>	6 7 3	X'FF'	==>	7 7 2

Figure 7-3 (Part 2 of 2). Converting KP1 from Hexadecimal to Keystroke Format

Verifying the Key

After entering a key into the keypad, you may later wish to verify that the key now in the keypad is the key you loaded. To do this, use the PIN keypad verification test that is supplied with the 4700 installation diskette.

You should perform this test in a secure environment.

You initiate the test by selecting option 5 on the installation diskette menu, as described in detail in *4700 Subsystem Operating Procedures.*

This test assumes that you have already loaded the protection key KP1 as described above. The test involves these steps, in order:

1. From the 4704 keyboard, enter KP1 in exactly the same format as it was originally entered in the keypad (24 keystrokes).

2. Enter an arbitrarily-chosen PIN via the 4704 keyboard. Enter at least one and not more than thirteen PIN characters. (You may wish to ensure that personnel who run this test use dummy PINs, not actual customer-assigned PINs.)

3. Enter the identical PIN via the PIN keypad.

The controller compares the two PINs and tells you whether they match. If they match, the keypad is operating correctly and the key in the keypad is the one you think it is. If the PINs do not match, the problem may be caused by any of the following:

- Your 4704-entered key and the keypad key are not the same.

- You didn't enter the two PINs in an identical manner.

- The keypad is defective (follow the problem reporting procedures described in *4704 Operating Instructions).*

How the Program and the Keypad Interact

The encrypting PIN keypad is a logical extension of the 4704 keyboard. Your program obtains its input from the keypad by first enabling the keypad and then issuing LREAD instructions.

The program enables the keypad by issuing a SIGNAL instruction. The SIGNAL instruction's parameter list must specify that *indicator light 3* is to be switched *on* (that is, X'21' bits set on). If you are not familiar with the SIGNAL and LREAD instructions, see *Volume 4: Loop and DCA Device Programming.*

The customer can enter a PIN as soon as the SIGNAL instruction has turned on the keypad indicator light. The customer enters a PIN consisting of one to thirteen keystrokes, then presses the keypad's END key. At any point before pressing the END key, the customer can press the ERASE key and start over. Pressing keys on the 4704 keyboard will not interrupt the operation.

At the completion of the LREAD instruction, the controller places the enciphered PIN in the program's segment storage. The enciphered PIN -- eKP1(PIN) -- is delimited with X'7E' and X'7F' characters.

The first four bits indicate how many PIN characters (keypad keystrokes) the customer entered. This value can range from X'1' (one-character PIN) to X'D' (thirteen-character PIN).

Each PIN character is represented in packed decimal form, four bits per character. The PIN can be as short as one character or as long as thirteen characters.

The variable-length PIN is padded with enough X'F' characters to make a thirteen-character field. The number of pad characters ranges from zero (for a thirteen-character PIN) to twelve (for a one-character PIN). Note that the total length of the LEN, PIN, and PAD fields is seven bytes.

When the customer enters a PIN, the encrypting PIN keypad appends a one-byte sequence number to the above, then enciphers the entire eight bytes. The keypad increments the sequence number each time a customer enters a PIN (that is, presses the END key). This makes the enciphered data more resistant to analysis, because the enciphered data is different each time -- even if the same PIN is entered several times in succession. The keypad starts at X'00' the first time power is applied to the keypad. After an END key has been pressed 256 times, the sequence number "rolls over" to X'00' again. If power is interrupted (keypad disconnected from the 4704, or the 4704 switched off), the sequence number resets to X'00' as soon as power is restored.

Chapter 8. Host Support Encryption Routines

The 4700 Host Support program (Licensed Program 5668-989) provides two encryption routines: BDKDPRS and BDKDES. BDKDPRS generates PINs and related data, while BDKDES is a DES encryption subroutine.

Host Support also contains a dummy routine BDKKEYCC, which you replace with your own program if you are using 4700's Host Support program to encipher a 3624 load image. BDKKEYCC is equivalent to BQKKEYCC, the routine used by 3600's Host Service programs for enciphering a 3624 load image. (Sections PR208 and PR209 of your *3624 Programmer's Reference* describe this routine.)

See *4700 Host Support User's Guide* for information about Host Support.

The BDKDPRS Routine

The BDKDPRS routine generates a PIN in plaintext if given a PIN validation key (KPv), a decimalization table, and validation data.

The routine can also be used to generate an offset value so that the actual PIN can be selected by the institution, rather than by BDKDPRS. In this second method of operation, BDKDPRS generates the proper offset data if given the same validation key (KPv), the same decimalization table, the same validation data, and the desired PIN.

If you are a 3624 programmer, you are probably already familiar with BDKDPRS. This routine is, in fact, the same routine, BQKDPRS, that you use to generate PINs and offset data for use in a 3624. (BQKDPRS is supplied with the 3600 Host Service programs.) If you are not a 3624 programmer, you may wish to read Chapter 6 of the *3624 Programmer's Guide.* That chapter contains many illustrations and examples of the PIN validation algorithm that are not duplicated here.

The 3624 PIN Validation Algorithm

The 4700 PINVERIF instruction and BDKDPRS use the following PIN validation algorithm. If you are writing your own equivalent of BDKDPRS, these steps should form the basis of your own program.

1. Obtain *KPv,* a plaintext PIN validation key. If KPv is enciphered under KM3, obtain KM, generate KM3, and decipher KPv.

2. Obtain the *validation data,* pad the data to eight bytes (if necessary), and encipher the data under KPv.

3. Replace each digit of the enciphered validation data with the digit in the *decimalization table* whose displacement from the beginning of the table is the same as the value of the digit of enciphered validation data. For example:

```
Decimalization table displacement:     0 1 2 . . .
Decimalization table data:             9 8 7 . . .
Original enciphered validation data:   1 2 2 . . .
Modified enciphered validation data:   8 7 7 . . .
```

4. Save the leftmost x digits as the *intermediate PIN,* where x is the number of digits in the customer-entered PIN. If your institution is selecting the PIN, proceed no further; the intermediate PIN is the PIN you assign to the customer. If your institution is allowing the customer to select the PIN, proceed to the next step.

5. You must now determine the *offset data* which, when added to the customer-selected PIN, yields the intermediate PIN determined in Step 4. To determine the offset data, subtract (modulo 10) the intermediate PIN from the customer-selected PIN.

6. Record the offset data where the program can obtain it (generally on the customer's identification card). If you are going to validate only part of the assigned PIN, record only the rightmost y digits of the offset data, where y is the number of PIN digits you are going to validate (the PIN check length).

   Note: Clear all intermediate PIN values and keys from storage before returning to the invoking program.

Generating PINs Without Offset Data

You provide BDKDPRS three data items -- *pinkey, dectab,* and *valdata,* two reserved work areas, *newpin* and *workarea,* and a parameter list of their addresses. The parameter list contains seven words, only five of which are used by BDKDPRS for new PIN generation. The remaining two (words 4 and 5) are used when generating offset data, as explained in the next section. However, word 4 must contain F'0' for the PIN generation process. You must ensure that parameters provided to BDKDPRS match those provided to the operational 4700 program.

BDKDPRS uses the following subroutine linkage conventions.

Register 1: Points to seven-word parameter list:

Word 0: Address of an eight-byte *pinkey* (KPv, the PIN validation key)

Word 1: Address of an eight-byte *dectab* (the decimalization table)

Word 2: Address of an eight-byte *valdata* (the validation data)

Word 3: Address of an eight-byte area reserved for *newpin*

Word 4: F'0'

Word 5: Reserved

Word 6: Address of 60-word work area reserved for *workarea*

Register 13: Contains address of calling routine's 18-word save area

Register 14: Contains calling routine's return address

Register 15: Contains address of BDKDPRS

valdata must be composed of the institution-defined validation data, padded if necessary to fill a field of sixteen characters. The validation data is left-justified in *valdata.* The padded *valdata* must be provided to the operational 4700 program for use in PINVERIF's validation data field.

BDKDPRS calls BDKDES to encipher the data in *valdata,* using the eight-byte key supplied in *pinkey. Pinkey* must be a PIN validation key, KPv.

BDKDPRS converts the enciphered validation data to decimal using the decimalization table stored in *dectab.* The decimalization table must be provided to the 4700 program for use in its PINVERIF decimalization table field. BDKDPRS stores the eight-byte decimalized result in the *newpin* field.

Because word 4 of the parameter list contains F'0', no offset calculation is performed.

The PIN check number must be the right-most portion of the PIN assigned to the customer. If the assigned PIN is longer than the PIN check number, you can insert any digits as the leading digits of the PIN.

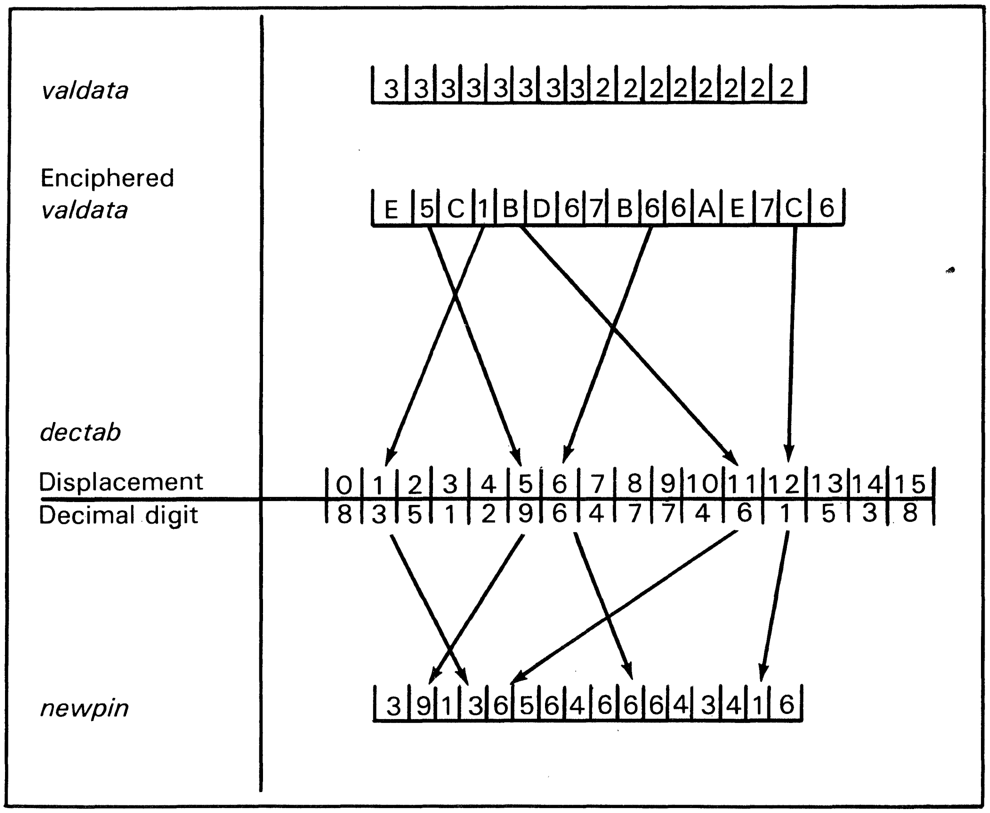
Calculating Newpin

Using the data provided in *valdata,* BDKDPRS calculates a 16-digit, packed decimal number and places it in *newpin.* BDKDPRS accepts any hexadecimal digits in *valdata.* It uses *pinkey* for enciphering *valdata,* and *dectab* for decimalizing the enciphered *valdata.*

BDKDPRS calculates *newpin* by mapping each of the hexadecimal digits of the enciphered *valdata* into a decimal digit of *dectab.* The decimal digit placed in *newpin* is that digit whose displacement (0 through 15) in *dectab* corresponds to the value (X'0' through X'F') of the hexadecimal digit being mapped.

In the following example:

Validation data = 33333333
Pad character = 2
pinkey = 89B07B35A1B3F47E



Assigning the PIN

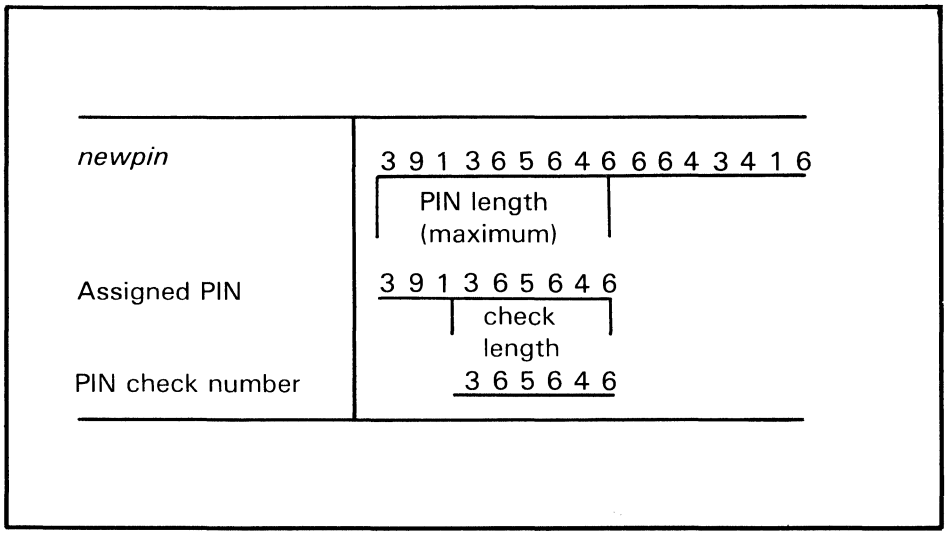
Using the data in *newpin,* the institution's program creates a PIN to be assigned to the customer whose validation data was used to calculate the value in *newpin.*

newpin is sixteen digits long. However, you need not assign the entire sixteen digits as the PIN. If you choose to assign fewer than sixteen digits, assign the leftmost y digits of *newpin* (where y is the number of digits you choose to assign).

Furthermore, you need not validate the entire PIN that you do assign -- this means that the customer does not have to enter the entire assigned PIN. You can select a *PIN check length* that is smaller, as shown in the next example. The PIN check length indicates the minimum number of PIN digits that the customer can enter. If you select a PIN check length that is smaller than the entered PIN, it is the *rightmost* portion of the PIN that is validated.

In the following example:

Assigned PIN length = 9
PIN check length = 6
newpin = 3913656466643416
Maximum PIN length = 9



The assigned PIN can be xxx365646, where x is any decimal digit, because the PIN validation algorithm checks only the rightmost six digits (the PIN check length) of the entered PIN.

You can also allow the customer to enter a variable number of digits of the assigned PIN. In this case, the assigned PIN must be 391365646. Because the algorithm checks only the rightmost six digits of the entered PIN, the customer can enter:

391365
x913656
xx136564
xxx365646

Summary of Institution Responsibilities

In summary, your responsibilities in programming the generation of new PINs are:

- Supply the PIN validation key, KPv, for enciphering *valdata.*
- Supply the decimalization table in *dectab.*
- Supply the customer's *validation data,* padded if necessary, in *valdata.*
- Ensure that data supplied to BDKDPRS is provided to the operational 4700 program (KPv must be supplied enciphered under KM3, however).
- Supply an eight-byte field for *newpin* and a 60-word *work area.*
- Call BDKDPRS, using subroutine linkage conventions described above.
- Use the contents of *newpin* to assign a PIN to the customer.

Generating PINs With Offset Data

In general, the institution's program collects data required to make up the parameter list used when calling the subroutine BDKDPRS, and uses the output from BDKDPRS for the final steps in generating offset data. It is your responsibility to ensure that data used in this routine is coordinated with the operational 4700 program.

You must provide BDKDPRS four data items, *pinkey, dectab, valdata,* and *oldpin,* three work areas, *newpin, offset,* and *workarea,* and a parameter list of their addresses. BDKDPRS uses the subroutine linkage conventions described above; only the use of the parameter list differs:

Word 0: Address of an eight-byte *pinkey* (for KPv, the PIN validation key)

Word 1: Address of an eight-byte *dectab* (the decimalization table)

Word 2: Address of an eight-byte *valdata* (the validation data)

Word 3: Address of an eight-byte area reserved for *newpin* (intermediate PIN)

Word 4: Address of an eight-byte *oldpin* (the desired PIN)

Word 5: Address of an eight-byte area reserved for *offset* data

Word 6: Address of 60-word area reserved for *workarea*

BDKDPRS calls BDKDES to encipher the data in *valdata,* using the eight-byte key supplied in *pinkey.* This key is the PIN verification key, KPv.

BDKDPRS converts the enciphered validation data to decimal, using the decimalization table stored in *dectab.* BDKDPRS stores the result in *newpin.* You need not use *newpin.*

BDKDPRS subtracts (modulo 10) the contents of *newpin* from the contents of the 16-digit field called *oldpin* and places the resulting difference in the field reserved for *offset. oldpin* must contain the PIN check number that was assigned to the customer.

The institution's program regains control from BDKDPRS and selects a number, beginning with the left-most digit of *offset,* whose length is the same as the length of the PIN assigned to the customer. Then, beginning with the right-most digit of the number just selected, and moving right to left, the program selects a number whose length is determined by the check length (that is, by the length of the PIN check number). The last number selected is the number that can now be recorded on the customer's card (or elsewhere) as the offset data.

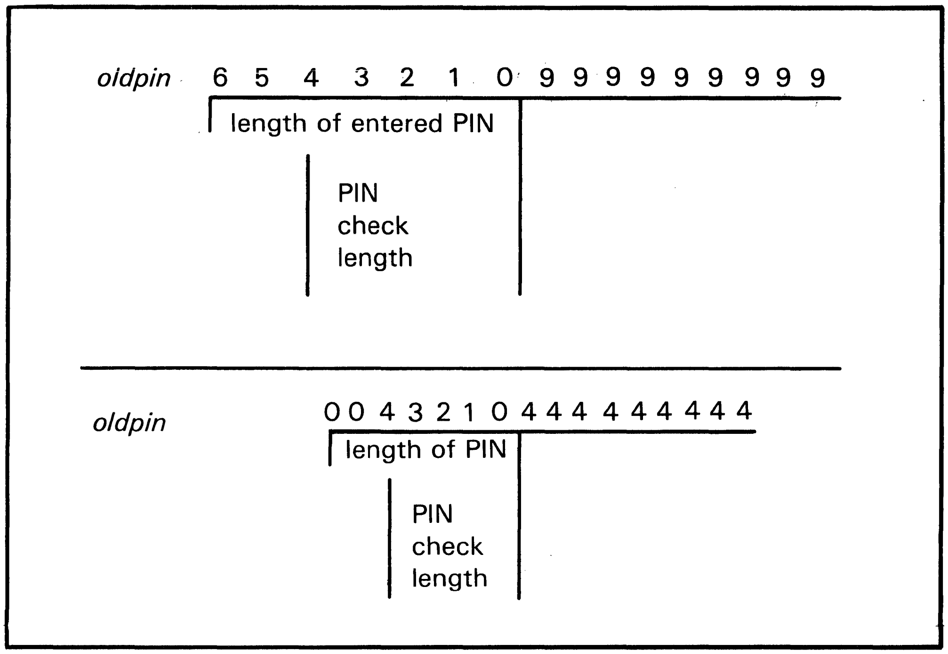
Constructing Oldpin

You are responsible for constructing *oldpin* so that it contains the customer's PIN check number in its proper position within *oldpin*. The PIN check number is right-justified in the assigned PIN.

In the following example, the entire PIN, including the PIN check number, is stored left-justified in *oldpin*.

In this example:

Assigned PIN = 6543210
Check length = 5
PIN check number = 43210



In the above illustration, the entire PIN is not used. However, the PIN check number must be used and it must be placed so that it falls in the same location within *oldpin* as it would if the entire PIN were used.

Calculating the Offset Data

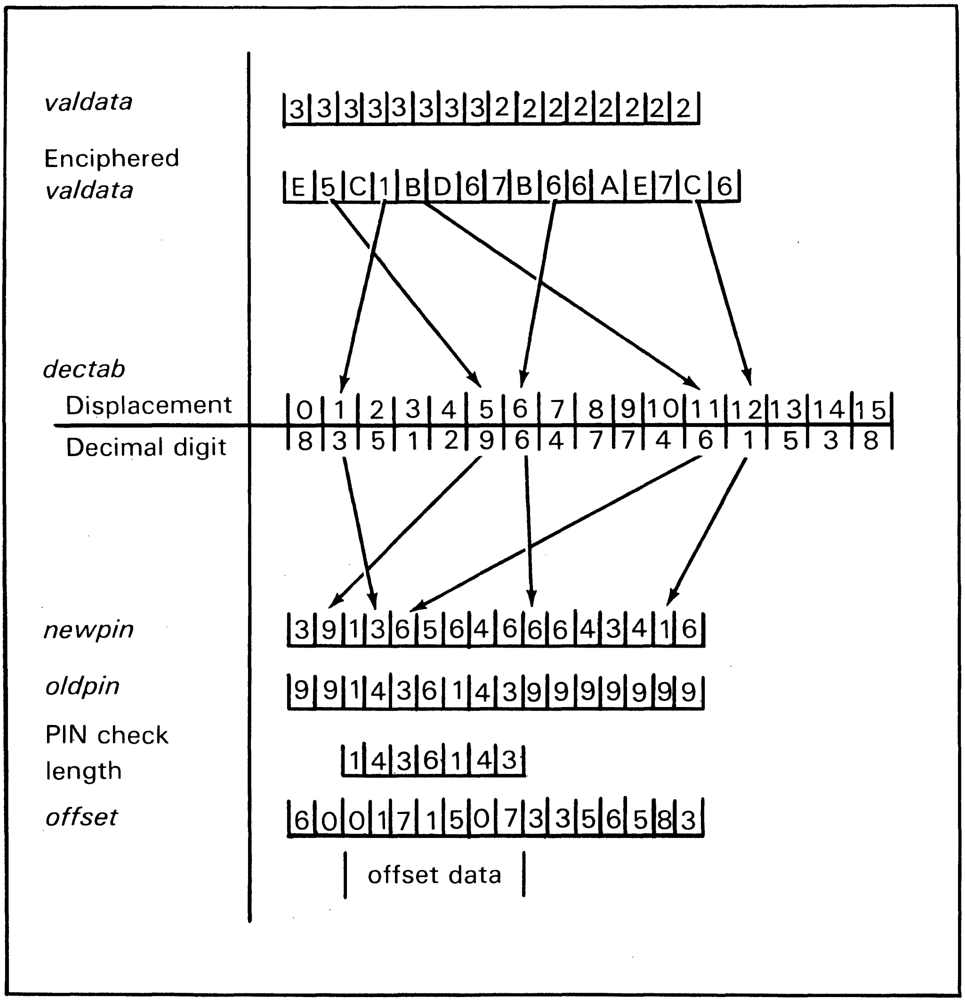
Using the data provided in *valdata*, BDKDPRS calculates a 16-digit packed-decimal number and places it in *newpin*. Then BDKDPRS subtracts, modulo 10, *newpin* from *oldpin*, and places the difference in *offset*.

For BDKDPRS, any hexadecimal digits are acceptable in *valdata*. The contents of *pinkey* are used for enciphering *valdata*, and the 16-packed-decimal digits in *dectab* are used to decimalize the enciphered *valdata*.

BDKDPRS calculates *newpin* by mapping each of the hexadecimal digits of the enciphered *valdata* into a decimal digit of *dectab*. The decimal digit placed in *newpin* is that digit whose displacement (0 through 15) in *dectab* corresponds to the value (X'0' through X'F') of the hexadecimal digit being mapped.

oldpin must be constructed as described above. BDKDPRS subtracts the contents of *newpin* from the contents of *oldpin*, digit by digit, with no carry. BDKDPRS places the difference in *offset*, as a 16-digit, packed-decimal number from which you select the offset data. In the following example, only the PIN check number is stored in *oldpin:*

Validation data = 33333333
Pad character = 2
pinkey = 89B07B35A1B3F47E
Assigned PIN = 361436143
PIN check number = 1436143
Check length = 7
oldpin = 9914361439999999



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

Summary of Institution Responsibilities

You have the following responsibilities when generating PIN offset data:

- Supply the key for enciphering *valdata.*
- Supply the decimalization table for *dectab.*
- Supply the customer's validation data, padded if necessary, in *valdata.*

- Supply the customer's assigned PIN check number in *oldpin*.
- Ensure that the data supplied to BDKDPRS is also available to the operational 4700 program (except for KPv, which must be supplied enciphered under KM3).
- Supply an eight-byte field for *newpin*, an eight-byte field for *offset*, and a 60-word *workarea*.
- Call BDKDPRS, using the linkage conventions described above.

The BDKDES Subroutine

BDKDES is another routine provided as part of the 4700 Host Support program. BDKDES is called by the BDKDPRS routine described above. BDKDES must also be present when you use the 4700 Host Support program to create an enciphered 3624 load image. (Section PR210 of the *3624 Programmer's Reference* describes BQKDES in this context.) BDKDES is equivalent to the BQKDES subroutine provided with the 3600 Host Service programs.

Your host program can also invoke this subroutine directly when you need to encipher or decipher data using plaintext keys. BDKDES performs encryption in the host computer that is analogous to ENCODE and DECODE encryption in the 4700 controller.

BDKDES Input

The host program must use the following linkage conventions to call BDKDES:

Register 1: Contains the address of the following parameter list:

Word 0: Request code: 0 indicates that the text is to be enciphered; any other value indicates that the text is to be deciphered

Word 1: Address of an eight-byte key used to encipher or decipher the data

Word 2: Address of the eight bytes of text to be enciphered or deciphered

Word 3: Address of a 30-word work area.

Register 13: Contains address of calling routine's 18-word save area

Register 14: Contains calling routine's return address

Register 15: Contains address of BDKDES

BDKDES Output

The subroutine replaces the input text pointed to by Word 2 with the enciphered or deciphered text.

The subroutine does not return completion codes or issue any messages. It treats all input values as valid. When the processing is complete, BDKDES restores the original contents of the registers. BDKDES does not change the four words in the parameter list.

The BDKKEYCC Subroutine

BDKKEYCC is a program you must write if you are using the Host Support program to transmit an enciphered 3624 load image. The purpose of the program is simply to provide Host Support the key under which it enciphers the load image. Host Support is shipped with a dummy BDKKEYCC subroutine which you replace.

The *4700 Host Support User's Guide* explains how you transmit an enciphered 3624 load image. Sections PR208 and PR209 of the *3624 Programmer's Reference* explain how you write the BDKKEYCC replacement.

Chapter 9. Maintaining Compatibility Between Different Levels of Cryptography

In the previous chapters, the controller program and the host program were assumed to be operating at an equivalent level of cryptography -- namely, 4700-level cryptography.

There are, of course, these other possibilities:

1. Your host computer is a System/34, or a System/370 operating with VSE (and you are therefore restricted to *3600-level cryptography)*, and you do not plan to upgrade your host processor or programming. To maintain compatibility with the 4700 subsystem, you can develop a 4700 program (or migrate a 3600 program) that uses only 3600-level cryptography. See "Implementing 3600-Level Cryptography in a 4700 Controller" below.

2. Your host computer uses *4700-level cryptography* (OS/VS cryptographic subsystem), and you plan to implement 3600-level cryptography in the 4700 controller. To maintain compatibility with the 3600-level controller program, you can write your host program in such a way that it emulates 3600-level cryptography. See "Implementing 3600-Level Cryptography with the OS/VS Cryptographic Subsystem" on page 9-2.

3. Your host computer uses *3600-level cryptography*, and you plan to upgrade your host programming. You can add instructions to your host programs so they emulate some of the functions of 4700-level cryptography. See "Implementing 4700-Level Cryptography in Host Computers" on page 9-3.

These options are summarized in Figure 9-1 on page 9-4 at the end of this chapter.

Implementing 3600-Level Cryptography in a 4700 Controller

3600-level cryptography is provided by the 4700 ENCODE and DECODE instructions. The support for these instructions is contained in the P57 module of controller data.

With one exception (described below), these instructions are identical to the ENCODE and DECODE instructions used to support 3600 controllers. These instructions encipher or decipher data...

- Using the DES algorithm
- Without cipher block chaining
- In fixed, eight-byte blocks, one block per instruction
- Without any provision for automatic padding
- Using a plaintext key supplied by the program.

You can use the 4700-level ENCIPHER and DECIPHER instructions to emulate the 3600-level ENCODE and DECODE instructions by enciphering and deciphering only eight bytes at a time with an initial chaining value of zero. Using this technique, you can use enciphered keys and still maintain compatibility with 3600-level cryptography.

Migrating 3600 Instructions

The only difference between the 3600 and 4700 versions of ENCODE and DECODE is that the 4700 version does not permit the use of the "alternate encryption technique" -- an algorithm supported by 3600's P56 module that is different from the DES (the US National Bureau of Standards' name for the data encryption algorithm).

The 3600 ENCODE and DECODE instructions therefore execute properly on 4700 controllers only if they specify *DES* encryption, rather than the alternate encryption technique. If your 3600 program uses the alternate encryption technique, you will have to modify your program so that it will operate with DES encryption. If your 3600 program uses DES encryption, no program changes are required.

The 4700 ENCODE and DECODE Instructions

The 4700 ENCODE instruction requires two inputs: the location of an eight-byte block of (presumed) plaintext, and an eight-byte, plaintext key. ENCODE enciphers the data under the key with the DEA and places the result into the same storage area that held the original plaintext. DECODE deciphers an eight-byte block of (presumed) ciphertext in a manner complementary to ENCODE.

```
• Data
• Key         ==> ENCODE ==> • eKey(Data)

• eKey(Data)  ==> DECODE ==> • Data
• Key
```

See Chapter 10, "4700 Cryptographic Instructions" for detailed descriptions of the ENCODE and DECODE instructions.

Implementing 3600-Level Cryptography with the OS/VS Cryptographic Subsystem

You can cause the CIPHER macro instruction to operate in a manner identical to the ENCODE or DECODE instructions by limiting the data length to eight and setting the initial chaining value to zero (which nullifies the effect of cipher block chaining). The plaintext key used with ENCODE and DECODE must be enciphered under the host master key, a task that can be accomplished with the EMK macro.

You can encipher a data item longer than eight bytes by repeatedly executing the CIPHER macro while holding the key constant, the ICV equal to zero, and increasing the CLERTXT and CPHRTXT parameters by eight for each execution. Unless the length of the data item is divisible by eight, padding must be performed prior to encryption. Remember that repeated executions of the CIPHER macro will degrade processor performance.

The syntax of the CIPHER macro instruction along with an explanation of the return codes can be found in the OS/VS Cryptographic Subsystem publications identified in the Bibliography.

Implementing 4700-Level Cryptography in Host Computers

4700-Level Cryptography with VSE

The 4700 Host Support program (5668-989) can be ordered with the encryption module, BDKDES. Host Support can be installed in both OS/VS and VSE operating systems. Host application programs can use BDKDES as a subroutine to encipher and decipher data. Chapter 8, "Host Support Encryption Routines" describes BDKDES and explains how to invoke it.

BDKDES enciphers and deciphers fixed, eight-byte blocks of data (cipher block chaining is not performed) based on a supplied plaintext key; BDKDES operation is equivalent to ENCODE and DECODE.

You can use BDKDES to emulate the 4700 ENCIPHER and DECIPHER operations provided you write your program to perform cipher block chaining and padding as specified by the 4700. This program could then be used as a subroutine by other programs--for example, to generate authentication codes.

Successful emulation of the 4700 ENCIPHER operation also provides an emulation of the 4700 MACGEN operation. Recall that the 4700 defines a MAC as the first four bytes of the last eight bytes of the ciphertext generated on some plaintext. Therefore, enciphering data also constitutes MAC generation--the additional task of the program is simply to know where in the ciphertext the MAC is located.

Again, remember that repeated executions of BDKDES have an adverse effect on processor performance. Exactly how much depends upon the amount of data being enciphered.

4700-Level Cryptography with System/34

The Finance Subsystem (feature 6010, 6011) of the System/34 operating system (program number 5726-SS1 with feature 6001 or 6002 and the prerequisite Interactive Communication Feature) provides several encryption subroutines. These subroutines are SUBR30 for RPGII, SUBR31 for COBOL, and #SBDE for assembler. SUBR30 and SUBR31 encipher up to 32 eight-byte blocks of data per call; #SBDE (like BDKDES) enciphers only one eight-byte block per call. All three routines use the data encryption algorithm without cipher block chaining and using plaintext keys.

You can use these subroutines to emulate the 4700 ENCIPHER and DECIPHER instructions. Note though that even though SUBR30 and SUBR31 can encipher more than one eight-byte block at a time, your own cipher block chaining program has to invoke these routines with only one eight-byte block at a time.

Processor performance is degraded in proportion to the number of times these routines are executed.

Cryptography with 8100 Computers

The IBM 8100 Information Processing System does not offer integrated cryptographic facilities.

If you do not install a 4700 controller at the 8100 site, 8100 application programs can relay enciphered data from one location to another.

An 8100 application program can also be written to examine data transmitted in plaintext. In this fashion data secrecy is sacrificed for the convenience of being able to interpret and take appropriate actions upon the content of the data without the opportunity to change or replace it.

Note that if the plaintext is protected with a message authentication code, however, the 8100 program cannot determine the plaintext's authenticity (the program has no MACGEN equivalent). Without this capability, the program might not be able to detect altered messages.

To maintain compatibility between your host facility (shown below)...	...And your 4700 Controller that implements...	
	• ENCODE and DECODE (3600-level) Cryptography	• ENCIPHER and DECIPHER (4700-level) Cryptography
OS/VS	Use Cryptographic Subsystem* with cipher block chaining disabled ...or use BDKDES.	Use Cryptographic Subsystem.*
VSE	Use BDKDES	Build a host program that uses BDKDES as a subroutine loop.
System/34	Use Finance Subsystem (SUBR30 for RPGII, SUBR31 for COBOL, or #SBDE for Assembler).	Build a host program that uses SUBR30, SUBR31, or #SBDE as a subroutine loop.
8100	Not applicable; relay enciphered data between 4700 and host computer.	Not applicable; relay enciphered data between 4700 and host computer.

*Programmed Cryptographic Facility, or combination of IBM 3848 Cryptographic Unit and the Cryptographic Unit Support Program

Figure 9-1. Maintaining Compatibility between Host and Controller Cryptographic Programs.

Chapter 10. 4700 Cryptographic Instructions

This chapter tells you how to use each of the 4700 cryptographic instructions:

DECIPHER	Decipher Text (4700-level)
DECODE	Decipher Text (3600-level)
ENCIPHER	Encipher Text (4700-level)
ENCODE	Encipher Text (3600-level)
KEYGEN	Generate Cryptographic Key
MACGEN	Generate Message Authentication Code (MAC)
PINTRANS	Translate Personal Identification Number (PIN)
PINVERIF	Validate Personal Identification Number (PIN)
RFMK	Reencipher From Master Key
RTMK	Reencipher To Master Key

Each instruction description tells you:

- what the instruction does
- how to set and interpret its parameter lists
- how to code the instruction
- what program checks and condition codes can result.

All of the instructions except ENCODE and DECODE require that you specify module P28 in the OPTMOD configuration macro (ENCODE and DECODE require module P57).

Syntax Notation Key

We use a uniform notation to describe the syntax of the controller symbolic instructions. The notation indicates which operands you must code and which are optional, the options that are available for expressing values, the values assumed by the system if you don't code an operand, and the punctuation.

CAPITAL LETTERS
Capital letters indicate values that you must enter exactly as shown.

lowercase letters
Lowercase letters indicate where you are to insert a number, character string, or keyword in place of the lowercase letters.

punctuation .,='()
The period, comma, equal sign, single quotation mark, and parentheses are punctuation that you must code exactly as shown. These punctuation marks separate the operands of the instructions. You need not code a comma preceding a keyword parameter for the *first* parameter in the operand field.

brackets []

Brackets indicate that you can choose not to code the elements and punctuation they enclose; the operand is *optional.*

braces {}

Braces indicate that you must code the elements and punctuation they enclose; the operand is *required.*

selecting options

When you choose from more than one operand, the choices appear like this, with vertical bars separating them:

```
[1|2|3]   or   {1|2|3}
```

or they might appear stacked, like this:

$$\begin{bmatrix}1\\2\\3\end{bmatrix} \quad \underline{\text{or}} \quad \begin{Bmatrix}1\\2\\3\end{Bmatrix}$$

In the following examples, the brackets indicate that you can choose not to code the *1*, *2*, or *3*, following *TYPE=element.*

$$\texttt{TYPE=element [1|2|3]} \quad \underline{\text{or}} \quad \texttt{TYPE=element} \begin{bmatrix}1\\2\\3\end{bmatrix}$$

But in the following example the braces indicate that you *must* code *1*, *2*, or *3*, following *TYPE=element.*

$$\texttt{TYPE=element \{1|2|3\}} \quad \underline{\text{or}} \quad \texttt{TYPE=element} \begin{Bmatrix}1\\2\\3\end{Bmatrix}$$

underscoring

We underscore a value to indicate that if you do not code a value for the element, the system *assumes* the underscored value. The value that the system assumes is called a *default.* In the following examples, if you do not code TYPE, the system uses *TYPE=1.*

$$[\texttt{TYPE=}\{\underline{1}|2|3\}] \quad \text{or} \quad \begin{bmatrix}\texttt{TYPE=}\begin{Bmatrix}1\\\underline{2}\\3\end{Bmatrix}\end{bmatrix}$$

ellipsis...

Ellipsis points indicate that you can add one or more additional operands or sets of operands, each having the same format. For example,

```
CASE=element 1 [,element 2,...element n]
```

indicates that you can repeat the syntactical unit (element) preceding the ellipsis.

DECIPHER--Decipher Text (4700-Level)

DECIPHER deciphers text using an enciphered key.

Program input to the DECIPHER instruction consists of the location of the ciphertext, the initial chaining value, an optional pad processing request, and the enciphered key. At the completion of the DECIPHER operation, the controller places the plaintext in the storage location that previously held the ciphertext.

```
• eK(Data)
• ICV            ==> DECIPHER ==> • Data
• eKM(K)
• pad processing
```

eK(Data): Place the ciphertext in any part of segment storage except Segment 14. The length of the ciphertext must be an exact multiple of eight, with a maximum length of 4096 bytes. Note that this "multiple of eight" requirement is different than the length requirement for ENCIPHER (where the data length does not have to be a multiple of eight bytes).

ICV: Provide the same eight-byte initial chaining value that was used to encipher the data.

Enciphered Key: Provide eKM(K), the same enciphered key that was used to encipher the data. This key must be enciphered under the controller's master key.

Padding: Indicate whether the controller is to perform pad processing. If you select this option, the controller verifies the pad count at the end of the deciphered data (must be from one to eight) and subtracts that number from the ENCLEN (length) field of the DECIPHER parameter list.

You use the ENCPAR parameter list to supply all input to DECIPHER. The structure of the parameter list for DECIPHER is the same as for ENCIPHER, and the same program storage can be used for both.

There are two differences in the way the controller uses ENCPAR for DECIPHER, however:

- the pad character field (ENCPADC) is ignored.

- if you specify padding, the controller sets the data length field (ENCLEN) upon completion of the DECIPHER operation.

DECIPHER | •

Parameter List (ENCPAR)

	Field	Name	Length	
====>	• Flag byte (1)	ENCFLG	1	
	• Pad character(2)	ENCPADC	1	
====>	• Key: eKM(K)	ENCKEY	8	
====>	• Initial chaining value	ENCIV	8	
	• Reserved		1	
====>	• Segment,(3)	ENCSEG	1	
====>	• displacement,	ENCDISP	2	
====>	• and length (4) of Data	ENCLEN	2	<==

(1)Set the X'80' bit on to indicate padding.
(2)Not used for DECIPHER.
(3)Relative to the logical work station's current space at DECIPHER execution time.
(4)Controller resets length if ENCFLG indicates pad processing.

Legend:
====> Program sets this field (always).
<== Controller sets this field (optional).

Use the COPY DEFENC instruction to provide addressing for the ENCPAR parameter list. The assembler labels and corresponding displacements and values generated by COPY DEFENC are described in Appendix B, "COPY File Fields."

The DECIPHER instruction has the following assembler format.

Name	Operation	Operand
[label]	DECIPHER	{ defld2 seg2,disp2 (reg2) (defrf2) }

operand 2

Is the parameter list for this instruction. The parameter list cannot be in segment 14. No length is specified because the parameter list has a fixed length of 24 bytes.

Condition Codes: If no program checks occur, DECIPHER returns one of the following hexadecimal codes in SMSCCD.

X'01'
: The instruction completed successfully; SMSDST has been set to zero.

X'02'
: The instruction did not complete successfully; a status code in SMSDST explains why.

Program Checks: One of the following hexadecimal codes can be set: 01, 02, 03, 09, or 27.

DECODE--Decipher Text (3600-Level)

Decode uses the DES algorithm to decipher 8 bytes of data to its original form. The data is deciphered in place under control of a specified key, which you supply in unenciphered form. After the deciphering occurs, the primary field pointer is increased by 8; thus, subsequent DECODE instructions can be used to decipher a field of any length in 8-byte increments.

```
Name      Operation  Operand

                          ⎧ defcon2      ⎫
                          ⎪ defld2       ⎪
[label] DECODE    seg1,   ⎨ (defrf2)     ⎬ , DES
                          ⎪ (reg2)       ⎪
                          ⎩ seg2,disp2   ⎭
```

operand 1

Defines the segment containing the field to be deciphered (do not use segment 14). The field's location is indicated by the primary field pointer (PFP) and the length is ignored. Following execution of the instruction, the PFP is increased by 8, and the field length indicator is set to 0.

operand 2

Defines the key to be used during deciphering. The length associated with this operand is ignored, and the first 8 bytes are assumed to be the key.

DES

Is a required operand that specifies the decryption algorithm; it must be specified.

Condition Codes: The code is not changed.

Program Checks: Any of the following hexadecimal codes can be set: 01, 02, 09, or 27.

ENCIPHER--Encipher Text (4700-Level)

ENCIPHER enciphers text, using a key that the program has enciphered under the controller master key.

Program input to the ENCIPHER instruction consists of the location and length of the data to be enciphered, an initial chaining value (for cipher block chaining), a pad character (optional), and the enciphered key.

At the completion of the ENCIPHER operation, the controller places the ciphertext, eK(Data), in the storage location that previously held the plaintext.

```
• Data
• ICV        ==> ENCIPHER ==> • eK(Data)
• eKM(K)
• {pad}
```

Data: Place the data to be enciphered in any part of segment storage except Segment 14. The length of the data *area* must be an exact multiple of eight, with a maximum length of 4096. The length of the *data* to be enciphered need not be a multiple of eight bytes if you specify padding.

ICV: Provide an arbitrary eight-byte initial chaining value. Any value can be used. See *Example 4* ("Example 4. Initiating a Cryptographic Session" on page 4-8) for a method of making the ICV available to a remote program.

Enciphered Key: Provide eKM(K), a data-encrypting key that has been enciphered under the controller master key. You can generate such an enciphered key with the KEYGEN instruction.

Padding: If you indicate to the ENCIPHER instruction that padding is to occur, you supply a one-byte pad character via a parameter list (not in the data). The controller first adds one extra byte to your data for use as a count field. The controller then determines how many bytes are now needed to make an exact multiple of eight (this could be from zero to seven), inserts that number of pad characters between the end of your data and the one-byte count field, and sets the count field to *that number plus one.* The count field now reflects the total number of added bytes, and the data length indicator reflects the length of the padded data.

Here is an example of how an input data area would appear after padding, but before the data is enciphered. (This is also how the data would appear following a DECIPHER operation.)

```
Your data area:
        ┌──────────────────────────────────┐
        │                 |                │
        └──────────────────────────────────┘
        |<────────── 16 bytes ──────────>|
Your data (10 bytes)
        ┌──────────────────────────────────┐
        │T E S T   M S G|  X               │
        └──────────────────────────────────┘

You specify padding, with a pad character of X'5C' (asterisk):
        ┌──────────────────────────────────┐
        │T E S T   M S G|  X * * * * * 6   │
        └──────────────────────────────────┘
         X'06' in the count byte ───>| |<───
```

In this padding example, you would supply a length specification of 10. At the completion of the ENCIPHER operation, the controller would reset this specification to 16.

You use the ENCPAR parameter list to supply all input to the ENCIPHER instruction. The parameter list (whose address is the sole operand of ENCIPHER) must conform to the following structure.

```
┌──────────────┐
│ ENCIPHER | • │
└───────────┬──┘
            │
            └──> Parameter List (ENCPAR)
```

	Field	Name	Length	
====>	• Flag byte(1)	ENCFLG	1	
==>	• Pad character	ENCPADC	1	
====>	• Key: eKM(K)	ENCKEY	8	
====>	• Initial chaining value	ENCIV	8	
	• Reserved		1	
====>	• Segment,(2)	ENCSEG	1	
====>	• displacement,	ENCDISP	2	
====>	• and length (3) of Data to be enciphered	ENCLEN	2	<==

(1)Set the X'80' bit on to indicate padding; the controller uses the current content of ENCPADC as the pad character.
(2)Relative to the logical work station's current space at ENCIPHER execution time.
(3)Controller resets length if ENCFLG indicates pad processing.

Legend:
====> Program sets this field (always).
==> Program sets this field (optional).
<== Controller sets this field (optional).

The COPY DEFENC instruction provides a table of assembler labels and corresponding displacements and values (a DSECT) that you can use to build the parameter list. The "Name" column above matches the names generated by COPY DEFENC.

The instructions generated by COPY DEFENC are described in Appendix B, "COPY File Fields."

The ENCIPHER instruction has the following assembler format:

Name	Operation	Operand
[label]	ENCIPHER	defld2 seg2,disp2 (reg2) (defrf2)

operand 2

Is the parameter list for this instruction. The parameter list cannot be in segment 14. No length is specified because the parameter list has a fixed length of 24 bytes.

Condition Codes: If no program checks occur, ENCIPHER returns one of the following hexadecimal codes in SMSCCD.

X'01'

The instruction completed successfully; SMSDST has been set to zero.

X'02'

The instruction did not complete successfully; a status code is present in SMSDST explaining why.

Program Checks: One of the following hexadecimal codes can be set: 01, 02, 03, 09, or 27.

ENCODE--Encipher Text (3600-Level)

ENCODE uses the DES algorithm to encipher 8 bytes of data. The data is enciphered in place under control of a key. After the enciphering occurs, the primary field pointer is increased by 8; thus, subsequent ENCODE instructions can be used to encipher a field of any length in 8-byte increments.

Name	Operation	Operand
[label]	ENCODE	seg1, { defcon2 defld2 (defrf2) (reg2) seg2,disp2 } , DES

operand 1

Defines the segment containing the field to be enciphered (do not use segment 14). The field's location is indicated by the primary field pointer (PFP) and the length is ignored. Following execution of the instruction, the PFP is increased by 8, and the field length indicator is set to 0.

operand 2

Defines the key to be used during enciphering. The length associated with this operand is ignored, and the first 8 bytes are assumed to be the key.

DES

Is a required operand specifying the DES encryption algorithm; it must be coded.

Condition Codes: The code is not changed.

Program Checks: Any of the following hexadecimal codes can be set: 01, 02, 09, or 27.

KEYGEN--Generate Cryptographic Key

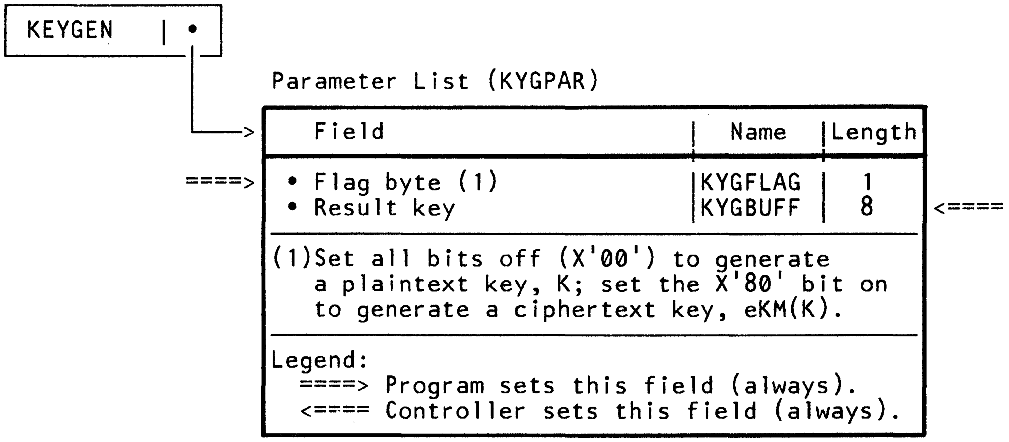
KEYGEN can be used in two ways:

- to create keys enciphered under the master key -- eKM(K) -- for use by programs during program execution
- to create plaintext keys -- K -- for use by security specialists

In both cases, the instruction generates a key that conforms in format with keys generated by other IBM cryptographic facilities. The plaintext key is generated with odd parity; the ciphertext key's parity bits are not set.

```
                    • eKM(K)
==> KEYGEN ==>        or
                    • K
```

You use the KYGPAR parameter list to supply all input to the KEYGEN instruction. The parameter list (whose address in the sole operand of KEYGEN) must conform to the following structure.



```
KEYGEN    | •
```

Parameter List (KYGPAR)

	Field	Name	Length	
====>	• Flag byte (1) • Result key	KYGFLAG KYGBUFF	1 8	<====

(1)Set all bits off (X'00') to generate a plaintext key, K; set the X'80' bit on to generate a ciphertext key, eKM(K).

Legend:
====> Program sets this field (always).
<==== Controller sets this field (always).

The parameter list contains the result key (in KYGBUFF) when the instruction completes successfully.

Note: The controller master key must be loaded prior to performing this instruction because the master key is used as one of the inputs for generating the resulting key.

The COPY DEFKYG instruction provides a table of assembler labels and corresponding displacements and values that you can use to build the parameter list. The *Name* column above matches the names generated by COPY DEFKYG.

The instructions generated by COPY DEFKYG are described in Appendix B, "COPY File Fields."

The KEYGEN instruction has the following assembler format.

Name	Operation	Operand
[label]	KEYGEN	defld2 seg2,disp2 (reg2) (defrf2)

operand 2

Is the parameter list for this instruction. The parameter list cannot be in segment 14. No length is specified because the parameter list has a fixed length of 9 bytes.

Condition Codes: If no program checks occur, KEYGEN returns one of the following hexadecimal codes in SMSCCD.

X'01'

The instruction completed successfully; SMSDST has been set to zero.

X'02'

The instruction did not complete successfully; a status code in SMSDST explains why.

Program Checks: One of the following hexadecimal codes can be set: 01, 02, 09, 11, or 27.

MACGEN--Generate Message Authentication Code (MAC)

The MACGEN instruction processes a data string and returns a four-byte message authentication code. You supply MACGEN a key and an initial chaining value just as though you were enciphering the data.

Internally, the controller enciphers a copy of the data using cipher block chaining. Unlike ENCIPHER, however, the controller does not return the entire enciphered message to you, nor does it disturb the input data string.

MACGEN operates like this:

```
• Data                         • Data (unchanged)
• eKM(KMAC) ==> MACGEN ==> • Last 8 bytes of
• ICV                             eKMAC(Data) - of which
• {pad}                                  first 4 are the MAC
```

Data: Place the data to be authenticated in any part of segment storage except Segment 14. The length of the data cannot exceed 4096 bytes. If you do not request padding, the data must be an exact multiple of eight bytes.

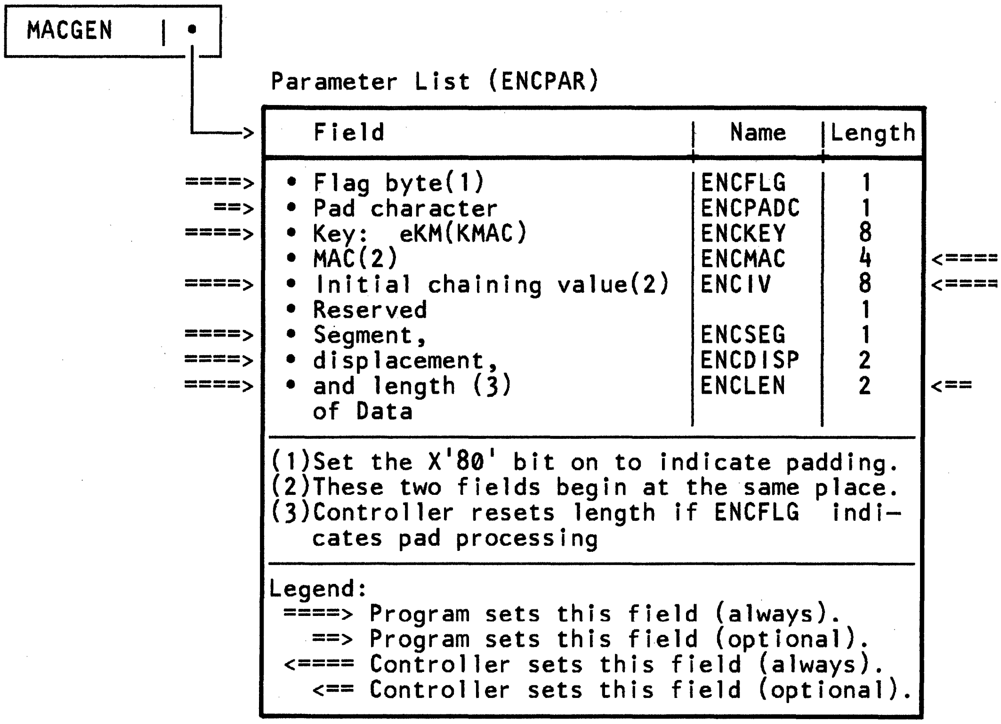
Key: Supply *KMAC,* a key that has been enciphered under the controller's master key (via the KEYGEN instruction, for example).

ICV: Provide an arbitrary eight-byte initial chaining value; any value can be used.

Padding: If you indicate that padding is to take place, you must further indicate what one-byte pad character is to be used. The controller adds a one-byte count character to the end of the input data, inserts enough pad characters (zero to seven) between the end of the data and the count character to make an exact multiple of eight, and sets the count character to the number of inserted pad characters plus one. The controller also resets your data length specification to reflect the length of the padded data. (Note that MACGEN's padding is identical to ENCIPHER's padding.)

After the controller has enciphered the entire data string -- eKMAC(Data) -- it returns only the final eight bytes to you. Use the *first four bytes* as the *message authentication code.*

You provide all MACGEN input via the ENCPAR parameter list — the same parameter list you use for the ENCIPHER and DECIPHER instructions. However, there is one difference in the way the controller handles the parameter list. For MACGEN, the eight-byte initial chaining value field serves as both an output and an input field. For multiple executions of MACGEN (as illustrated in Chapter 5, "Authenticating Messages"), the entire eight-byte field can serve as both an output field for one MACGEN and as an input field for the next. For single MACGEN operations (and for the last of a series of MACGEN operations), you take the MAC from the first half of this field.



Parameter List (ENCPAR)

	Field	Name	Length	
====>	• Flag byte(1)	ENCFLG	1	
==>	• Pad character	ENCPADC	1	
====>	• Key: eKM(KMAC)	ENCKEY	8	
	• MAC(2)	ENCMAC	4	<====
====>	• Initial chaining value(2)	ENCIV	8	<====
	• Reserved		1	
====>	• Segment,	ENCSEG	1	
====>	• displacement,	ENCDISP	2	
====>	• and length (3) of Data	ENCLEN	2	<==

(1)Set the X'80' bit on to indicate padding.
(2)These two fields begin at the same place.
(3)Controller resets length if ENCFLG indicates pad processing

Legend:
====> Program sets this field (always).
==> Program sets this field (optional).
<==== Controller sets this field (always).
<== Controller sets this field (optional).

When the instruction completes successfully, the controller places the combination four-byte MAC, eight-byte chaining value into the parameter list's ENCMAC/ENCIV field.

Use the COPY DEFENC instruction to provide addressing for the parameter list. The assembler labels and corresponding displacements and values generated by COPY DEFENC are described in Appendix B, "COPY File Fields."

The MACGEN instruction has the following assembler format.

Name	Operation	Operand
[label]	MACGEN	{ defld2 seg2,disp2 (reg2) (defrf2) }

operand 2

Is the parameter list for this instruction. The parameter list cannot be in segment 14. No length is specified because the parameter list has a fixed length of 24 bytes.

Condition Codes: If no program checks occur, MACGEN returns one of the following hexadecimal codes in SMSCCD.

X'01'

The instruction completed successfully; SMSDST has been set to zero.

X'02'

The instruction did not complete successfully; a status code is present in SMSDST explaining why.

Program Checks: One of the following hexadecimal codes can be set: 01, 02, 03, 09, or 27.

PINTRANS--Translate a Personal Identification Number (PIN)

PINTRANS converts a PIN from one format to another and/or reenciphers the PIN under a different PIN protection key. PINTRANS, like PINVERIF, processes enciphered PINs without revealing the PIN in plaintext.

PINTRANS accepts PINs in any of the following formats

- ANSI Standard PIN format (enciphered under KP1)
- Nonencrypting PIN keypad format
- Encrypting PIN keypad format (enciphered under KP1)
- 3624 PIN format (enciphered under KP1)
- 3621 PIN format (enciphered under KP1)

and produces the same PIN in any of the following formats:

- ANSI Standard PIN format
- Encrypting PIN keypad format
- 3624 PIN format
- 3621 PIN format.

PINTRANS operates in the following manner when the input PIN is in *nonencrypting* PIN keypad format or in *3624* PIN format:

```
• PIN
• eKM1(KP2) ==> PINTRANS ==> • eKP2(PIN)
```

PINTRANS operates in the following manner when the input PIN is in one of the *encrypting* PIN keypad formats:

Reenciphering Under the Same Key—

```
• eKP1(PIN)
• eKM3(KP1) ==> PINTRANS ==> • eKP1(PIN)
```

Reenciphering Under a New Key—

```
• eKP1(PIN)
• eKM3(KP1) ==> PINTRANS ==> • eKP2(PIN)
• eKM1(KP2)
```

The first operand of PINTRANS points to an input parameter list, INPPAR. This parameter list contains information about the input PIN, and must conform to the structure shown below. A second parameter list provides information about the output PIN. PINTRANS uses the same input parameter list as PINVERIF.

```
 ┌───────────────────┐
 │ PINTRANS | • | •─ │───────> Translation Parameter List
 └──────────────┬────┘
                │
                │     Input Parameter List (INPPAR)
                │    ┌──────────────────────────────────────────────┐
                └───>│ Field                        | Name   |Length│
                     ├──────────────────────────────┼────────┼──────┤
               ====> │ • Flag byte(1)               |INPINTYP|  1   │
                 ==> │ • Pad character(2)           |INPINPAD|  1   │
                 ==> │ • Protection Key(3)          |INPINKEY|  8   │
                 ==> │ • Primary Account Number (4) |INPAN   |  8   │
                 ==> │ • PIN length (5)             |INPINLEN|  1   │
               ====> │ • Enciphered PIN(6)          |INPIN   |  16  │<====
                     ├──────────────────────────────────────────────┤
                     │(1)Set to indicate type of input PIN format:  │
                     │    X'00' = Nonencrypting PIN keypad          │
                     │    X'10' = 3621                              │
                     │    X'20' = ANSI                              │
                     │    X'40' = 3624                              │
                     │    X'80' = Encrypting PIN keypad.            │
                     │(2)Used only if 3621 or 3624 format;          │
                     │   low-order 4 bits indicate pad character    │
                     │(3)eKM3(KP1); not used if format = X'00'.     │
                     │(4)Used only if format = X'20'                │
                     │(5)Used only if format = X'00'; indicates     │
                     │   number of EBCDIC PIN characters.           │
                     │(6)eKP1(PIN) (or the PIN, if format = X'00')  │
                     │   must be placed in the high-order end of    │
                     │   this field; the controller places the      │
                     │   resulting eKP1(PIN) or eKP2(PIN) in the    │
                     │   same high-order eight bytes of field.      │
                     ├──────────────────────────────────────────────┤
                     │Legend:                                       │
                     │  ====> Program sets this field (always).     │
                     │    ==> Program sets this field (optional).   │
                     │  <==== Controller sets this field (always).  │
                     └──────────────────────────────────────────────┘
```

Use the COPY DEFINP instruction to provide addressing for the INPPAR parameter list. The assembler labels and corresponding displacements and values generated by COPY DEFINP are described in Appendix B, "COPY File Fields."

The second operand of PINTRANS must point to a translation parameter list, TNPPAR. The translation parameter list contains information about the output PIN, and must conform to the following structure.

```
┌──────────────────────┐
│ PINTRANS | • | •─────┼──┐
└──────────────────────┘  V
            |      Translation Parameter List (TNPPAR)
            |     ┌─────────────────────────────────────────────┐
 Input <────┘     │  Field                  | Name    |Length   │
 Parameter        ├─────────────────────────┼─────────┼─────────┤
 List      ====>  │ • Flag byte(1)          |TNPINFLG |  1      │
           ====>  │ • Pad character(2)      |TNPINPAD |  1      │
                  │ • Reserved              |         |  2      │
             ==>  │ • Output PIN key(3)     |TNPINKEY |  8      │
             ==>  │ • Output PAN (4)        |TNPAN    |  8      │
                  ├─────────────────────────────────────────────┤
                  │(1)High order 4 bits set as follows:         │
                  │    X'1' = 3621 PIN format                   │
                  │    X'2' = ANSI Standard PIN format          │
                  │    X'4' = 3624 PIN format                   │
                  │    X'8' = Encrypting PIN format             │
                  │   Low order 4 bits set as follows:          │
                  │    X'0' to reencipher under KP2             │
                  │    X'1' to reencipher under KP1             │
                  │(2)Low order 4 bits indicate 3621 or 3624    │
                  │   PIN pad character (X'xP').                │
                  │(3)eKM1(KP2) if reenciphering PIN under KP2. │
                  │(4)Output Primary Account Number             │
                  ├─────────────────────────────────────────────┤
                  │Legend:                                      │
                  │  ====> Program sets this field (always).    │
                  │    ==> Program sets this field (optional).  │
                  └─────────────────────────────────────────────┘
```

The COPY DEFTNP instruction provides a table of assembler labels and corresponding displacements and values that you can use to build the translation parameter list. The *Name* column above matches the names generated by COPY DEFTNP.

The assembler instruction format for PINTRANS appears below. The first operand refers to the input parameter list (INPPAR) and the second refers to the translation parameter list (TNPPAR).

```
Name     Operation  Operand

                    ┌ defld1    ┐   ┌ defld2    ┐
[label]  PINTRANS   │ seg1,disp1│   │ seg2,disp2│
                    │ (reg1)    │   │ (reg2)    │
                    └ (defrf1)  ┘   └ (defrf2)  ┘
```

Operand 1
Is the input parameter list, INPPAR. The input parameter list cannot be in segment 14. The parameter list has a fixed length of 35 bytes.

Operand 2
Is the translation parameter list, TNPPAR. The translation parameter list has a fixed length of 20 bytes, and cannot reside in segment 14.

The high-order eight bytes of INPIN (in the input parameter list) contain the enciphered output PIN when PINTRANS completes successfully. PINTRANS also sets statistical counters as described in Appendix F, "Statistical Counters."

Condition Codes: If no program checks occur, PINTRANS returns one of the following hexadecimal codes in SMSCCD.

X'01'

The instruction completed successfully; SMSDST has been set to zero.

X'02'

The instruction did not complete successfully; a status code in SMSDST explains why.

Program Checks: One of the following codes can be set: 01, 02, 09, 11, or 27.

PINVERIF--Validate a Personal Identification Number (PIN)

The PINVERIF operation can be summarized as follows:

```
• PIN Input                           Check or
• Validation ==> PINVERIF ==> • No–Check
  Input                                 Indication
```

The PIN can be in any of the supported 4700 formats: nonencrypting PIN keypad, encrypting PIN keypad, ANSI, 3621, or 3624. The input parameter list (INPPAR) provides PINVERIF with the information related to the input *PIN.* The information related to the validation data is provided by a second parameter list.

```
┌──────────────────────┐
│ PINVERIF | • | •–│—> Verification Parameter List
└───────────┬──────────┘
            │      Input Parameter List (INPPAR)
            └—————>
```

	Field	Name	Length
====>	• Flag byte(1)	INPINTYP	1
==>	• Pad character(2)	INPINPAD	1
==>	• Protection Key(3)	INPINKEY	8
==>	• Input PAN (4)	INPAN	8
==>	• Check length(5)	INPINLEN	1
====>	• Enciphered PIN(6)	INPIN	16

(1)Set to indicate type of input PIN format:
X'00' = Nonencrypting PIN keypad
X'10' = 3621
X'20' = ANSI Standard PIN
X'40' = 3624
X'80' = Encrypting PIN keypad.
(2)Used only if 3621 or 3624 format; low–order 4 bits indicate pad character
(3)Contains eKM3(KP1); not used if format = X'00'.
(4)Input Primary Account Number
(5)Used only if format = X'00'; indicates number of EBCDIC PIN characters.
(6)Contains eKP1(PIN) (or the plaintext PIN, if format = X'00') in the high–order end of this field.

Legend:
====> Program sets this field (always).
==> Program sets this field (optional).

The COPY DEFINP instruction provides a table of assembler labels and corresponding displacements and values that you can use to address the parameter list. The *Name* column above matches the names generated by COPY DEFINP.

The instructions generated by COPY DEFINP are described in Appendix B, "COPY File Fields."

The following table (Figure 10-1) summarizes how you set the input parameter list, depending on the format of the PIN being validated:

INPPAR Field	Nonencrypting PIN Keypad	Encrypting PIN Keypad	3624 PIN Format	ANSI PIN Format	3621 PIN Format
INPINTYP	X'00'	X'80'	X'40'	X'20'	X'10'
INPINPAD	not used	not used	X'xP'	not used	X'xp'
INPINKEY	not used	eKM3(KP1)	eKM3(KP1)	eKM3(KP1)	eKM3(KP1)
INPAN	not used	not used	not used	Pri. Act. #	not used
INPINLEN	X'01' - X'10'	not used	not used	not used	not used
INPIN	PIN	eKP1(PIN)	eKP1(PIN)	eKP1(PIN)	eKP1(PIN)

Figure 10-1. Summary of INPPAR Settings for Each PIN Format

For Nonencrypting PIN Keypad

- INPINTYP must be set to X'00'.
- INPINLEN must contain a value from one to sixteen, representing the number of digits in the customer-entered PIN.
- INPIN must contain the customer-entered PIN in EBCDIC, as described above. The value must be left-justified in the field and contain the number of characters indicated by INPINLEN.

For Encrypting PIN Keypad

- INPINTYP must be set to X'80'.
- INPINKEY must contain eKm3(KP1), the input PIN protection key enciphered under the third variant of the controller master key.
- INPIN must contain the eight-byte enciphered PIN.

For 3624 PIN Format

- INPINTYP must be set to X'40'.
- INPINKEY must contain eKm3(KP1), the input PIN protection key enciphered under the third variant of the controller master key.
- INPIN must contain the eight-byte enciphered PIN.
- INPINPAD must be set to X'xP' where P is the four-bit hexadecimal digit that the 3624 adds to the PIN. (the high-order half of the byte -- x -- is ignored.)

Usually, you would have configured the 3624 to insert a pad character in the range of X'A' to X'F'. If the 3624 is inserting a pad character outside of this range (decimal digits 0 - 9, for example), you must ensure that no customer-entered PIN contains this digit. Otherwise, the controller considers that digit to be a pad character, and ignores all subsequent digits.

The second operand of the PINVERIF instruction points to a verification parameter list. This parameter list (VER3PAR) provides input for the verification algorithm.

```
┌──────────────────────┐
│ PINVERIF | • | •─────┼──┐
└──────────────────────┘  V
              │     Verification Parameter List (VER3PAR)
              │    ┌──────────────────────────────────────────┐
Input <───────┘    │   Field                 |  Name   |Length│
Parameter          ├─────────────────────────+─────────+──────┤
List       ====>   │ • Flag byte(1)          |V3ALGTYP |  1   │
                   │   Reserved              |         |  1   │
           ====>   │ • Validation key(2)     |V3PINKEY |  8   │
           ====>   │ • Decimalization table(3)|V3DECTAB|  8   │
           ====>   │ • Validation data(4)    |V3VALDAT |  8   │
           ====>   │ • Check length(5)       |V3CHKLEN |  1   │
           ====>   │ • Offset data(6)        |V3OFFSET |  8   │
                   ├──────────────────────────────────────────┤
                   │(1)Must be set to X'00'.                  │
                   │(2)Must be in enciphered form: eKM3(KPv). │
                   │(3)Contains 16 packed decimal digits.     │
                   │(4)Contains 1 to 16 4-bit validation      │
                   │   characters, padded on right.           │
                   │(5)Indicates number of PIN digits to be   │
                   │   compared with check number; from X'01' │
                   │   to X'10'.                              │
                   │(6)Contains 1 to 16 4-bit offset characters,│
                   │   X'0' to X'9', left-justified; if there is│
                   │   no offset data, set this field to zeros.│
                   ├──────────────────────────────────────────┤
                   │Legend:                                   │
                   │ ====> Program sets this field (always).  │
                   └──────────────────────────────────────────┘
```

The COPY DEFVER3 instruction provides a table of assembler labels and corresponding displacements and values that you can use to build the VER3PAR parameter list. The *Name* column above matches the names generated by COPY DEFVER3.

The instructions generated by COPY DEFVER3 are described in Appendix B, "COPY File Fields."

The assembler instruction format for PINVERIF is shown below. The first operand refers to the input parameter list (INPPAR) and the second refers to the verification parameter list (VER3PAR).

```
Name     Operation  Operand

                    ⎧ defld1       ⎫   ⎧ defld2       ⎫
[label]  PINVERIF   ⎨ seg1,disp1   ⎬ , ⎨ seg2,disp2   ⎬
                    ⎪ (reg1)       ⎪   ⎪ (reg2)       ⎪
                    ⎩ (defrf1)     ⎭   ⎩ (defrf2)     ⎭
```

Operand 1

Is the input parameter list, INPPAR. The input parameter list cannot be in segment 14. You specify no length, because the parameter list has a fixed length 35 bytes.

Operand 2

Is the verification parameter list, VER3PAR. The verification parameter list cannot be in segment 14. You specify no length, because the parameter list has a fixed length of 35 bytes.

Condition Codes: If no program checks occur, PINVERIF returns one of the following hexadecimal codes in SMSCCD.

X'01'

The instruction completed successfully; the PIN passed its validation check.

X'02'

Either the PIN failed its validation check, or the instruction did not complete successfully. If the instruction did not complete successfully, a status code in SMSDST explains why.

Program Checks: One of the following hexadecimal codes can be set: 01, 02, 09, 11, or 27.

RFMK--Reencipher From Master Key

The RFMK (Reencipher From Master Key) instruction takes a key that has been enciphered under the controller's master key (KM), deciphers it, and reenciphers it under a different key. The different key can be either KCD1 or eKM1(KCDs).

When you use RFMK to reencipher under *KCD1*, RFMK operates like this:

```
• eKM(K) ==> RFMK ==> • eKCD1(K)
```

When you use RFMK to reencipher under *KCDs*, RFMK operates like this:

```
• eKM(K)
• eKM1(KCDs) ==> RFMK ==> • eKCDs(K)
```

You provide all RFMK input in the form of a parameter list, whose address is the sole operand of the instruction. The parameter list must conform to the following structure.

```
┌───────────────┬───┐
│ RFMK          | • │
└───────────────┴─┬─┘
                  │      Parameter List (RMKPAR)
                  │     ┌──────────────────────────────────────────────────┐
                  └───> │    Field                     |  Name    |Length  │
                        ├──────────────────────────────┼──────────┼────────┤
                  ====> │  • Flag byte(1)              |RMKFLAG   |   1    │
                  ====> │  • Key being reenciphered(2) |RMKINKEY  |   8    │ <====
                    ==> │  • eKM1(KCDs)(3)             |RMKXDKEY  |   8    │
                        ├──────────────────────────────────────────────────┤
                        │ (1)Set all bits off (X'00') to reencipher to     │
                        │    KCD1; set the X'80' bit on to reencipher      │
                        │    to KCDs.                                      │
                        │ (2)As input, you set this field in the form      │
                        │    eKM(K); as output, the controller sets the    │
                        │    field in the form eKCD1(K) or eKCDs(K).       │
                        │ (3)Set this field only when you are              │
                        │    reenciphering K using KCDs — eKCDs(K).        │
                        ├──────────────────────────────────────────────────┤
                        │ Legend:                                          │
                        │   ====> Program sets this field (always).        │
                        │     ==> Program sets this field (optional).      │
                        │   <==== Controller sets this field (always).     │
                        └──────────────────────────────────────────────────┘
```

The parameter list's RMKINKEY field contains the reenciphered key when RFMK completes successfully.

The COPY DEFRMK instruction provides a table of assembler labels and corresponding displacements and values that you can use to build the RMKPAR parameter list. The *Name* column above matches the names generated by COPY DEFRMK.

The instructions generated by COPY DEFRMK are described in Appendix B, "COPY File Fields."

The RFMK instruction has the following assembler format.

Name	Operation	Operand

`[label]`	`RFMK`	`defld2`
		`seg2,disp2`
		`(reg2)`
		`(defrf2)`

operand 2

Is the parameter list for this instruction. The parameter list cannot be in segment 14. You specify no length, because the parameter list has a fixed length of 18 bytes.

Condition Codes: If no program checks occur, RFMK returns one of the following hexadecimal codes in SMSCCD.

X'01'

The instruction completed successfully; SMSDST has been set to zero.

X'02'

The instruction did not complete successfully; a status code in SMSDST explains why.

Program Checks: One of the following hexadecimal codes can be set: 01, 02, 09, 11, or 27.

RTMK--Reencipher To Master Key

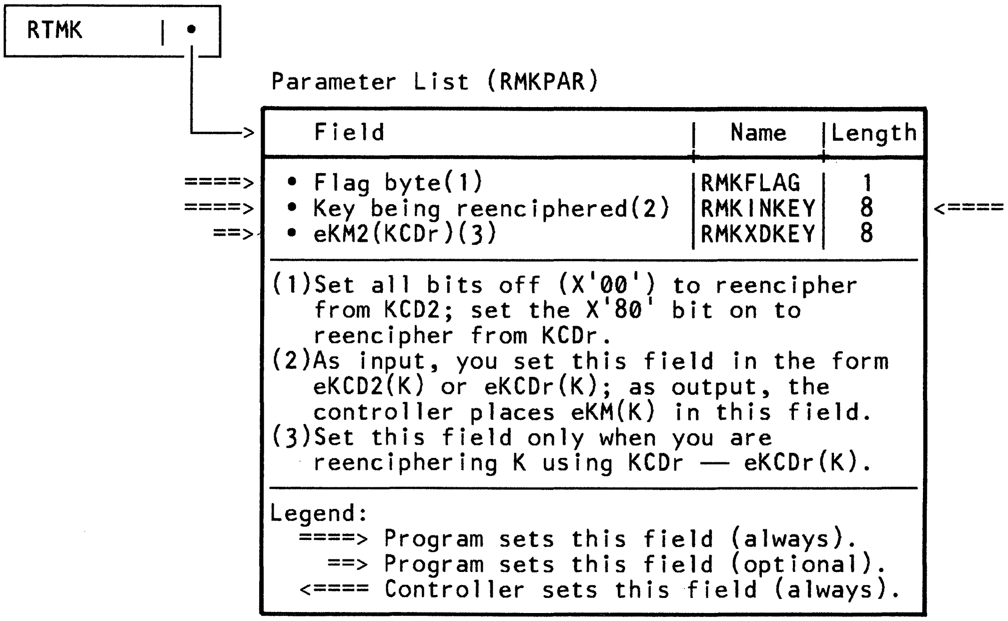
The RTMK (Reencipher To Master Key) instruction takes a key that has been enciphered under one KCD2 or eKM2(KCDr) and reenciphers it under the controller master key, KM. When you use RTMK to reencipher a key that was enciphered under *KCD2,* RTMK operates like this:

```
• eKCD2(K) ==> RTMK ==> • eKM(K)
```

When you use RTMK to reencipher a key that was enciphered under *KCDr,* RTMK operates like this:

```
• eKCDr(K)
• eKM2(KCDr) ==> RTMK ==> • eKM(K)
```

The RTMK instruction uses the RMKPAR parameter list — the same parameter list used by RFMK. Only the interpretation of the fields is different.



RTMK | •

Parameter List (RMKPAR)

	Field	Name	Length	
====>	• Flag byte(1)	RMKFLAG	1	
====>	• Key being reenciphered(2)	RMKINKEY	8	<====
==>	• eKM2(KCDr)(3)	RMKXDKEY	8	

(1)Set all bits off (X'00') to reencipher from KCD2; set the X'80' bit on to reencipher from KCDr.
(2)As input, you set this field in the form eKCD2(K) or eKCDr(K); as output, the controller places eKM(K) in this field.
(3)Set this field only when you are reenciphering K using KCDr — eKCDr(K).

Legend:
====> Program sets this field (always).
==> Program sets this field (optional).
<==== Controller sets this field (always).

You use the COPY DEFRMK instruction to provide addressing for the parameter list. The assembler labels and corresponding displacements and values generated by COPY DEFRMK are described in Appendix B, "COPY File Fields."

The RTMK instruction has the following assembler format.

Name	Operation	Operand
[label]	RTMK	defld2 seg2,disp2 (reg2) (defrf2)

operand 2

Is the parameter list for this instruction. The parameter list cannot be in segment 14. You specify no length, because the parameter list has a fixed length of 18 bytes.

Condition Codes: If no program checks occur, RTMK returns one of the following hexadecimal codes in SMSCCD.

X'01'

The instruction completed successfully; SMSDST has been set to zero.

X'02'

The instruction did not complete successfully; a status code in SMSDST explains why.

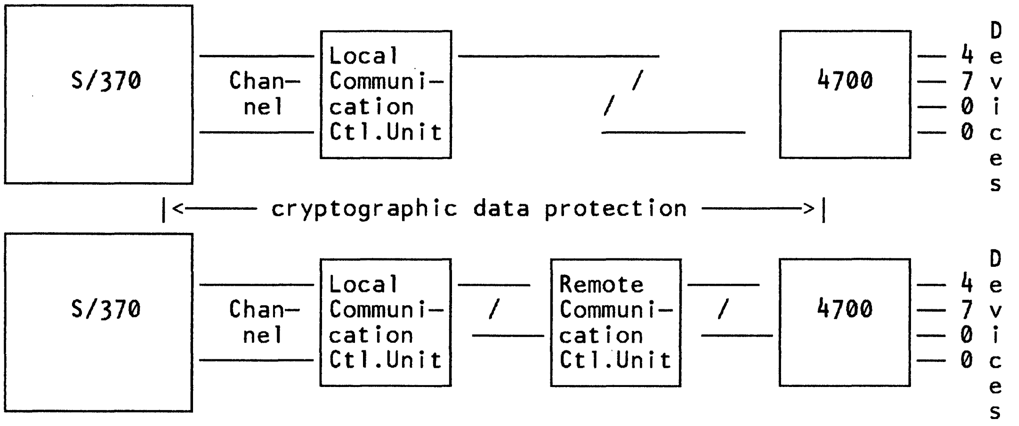
Program Checks: One of the following hexadecimal codes can be set: 01, 02, 09, 11, or 27.

Chapter 11. Guidelines for Coordinating 4700 and OS/VS Cryptographic Facilities

A high degree of communication security can be attained among application programs if they are properly designed to use cryptography.

Generally speaking, when both a S/370 and a 4700 provide cryptographic services for use by their respective programs, data is protected by cryptography before leaving the safety of either the S/370 host or the 4700, and remains enciphered— regardless of the number of intermediate network nodes through which it may pass— until deciphered after arriving at its intended destination. A PIN, for example, after being enciphered at an encrypting PIN keypad, remains enciphered during all subsequent periods of transmittal or storage until completing its role in the PIN validation process.

The following diagram illustrates two possible network configurations involving S/370 hosts and 4700 Finance Communication Systems.


S/370
Chan-nel
Local Communi-cation Ctl.Unit
4700
4700
Devices
|<—— cryptographic data protection ——>|
S/370
Chan-nel
Local Communi-cation Ctl.Unit
Remote Communi-cation Ctl.Unit
4700
4700
Devices

The following sections highlight the major concerns you must consider in your design of host and 4700 application programs that will use cryptography.

Communication Considerations

When the 4701 operates online (that is, in session with a host computer), it uses the facilities provided by a communication common carrier to transmit and receive data. Data exchanges may be *transaction* oriented, which is the choice for a typical retail banking application (consumer-operated cash issuing terminal, for example) or *message* oriented which is the choice for more conventional data processing applications such as the daily posting or reconciliation of a branch bank's records with those of the home office. Because the common carrier typically guarantees only delivery of data from source to destination, the use of protective measures to defend against possible wiretap attacks is left to the system or the appropriate application programs.

Having selected cryptography as a data security technique to achieve a high degree of communication security, you must then determine when, where, and how often it is to be used. The answer to these questions depends on the value of the data being transmitted, the frequency and volume of the data being transmitted, and the processing delays resulting from enciphering and deciphering operations. Similarly, you must determine the value, if any, of enciphering acknowledgments in terms of the protection realized versus the cost in terms of additional processing time. The answers to these questions ultimately depend on the general nature of the communication system and the means by which it satisfies the business requirements of the institution it supports.

Data Encryption with the OS/VS Cryptographic Subsystem

In OS/VS (OS/VS1 or OS/VS2 MVS) host systems that incorporate

- PCF (Programmed Cryptographic Facility, program number 5740-XY5), or
- CUSP (Cryptographic Unit Support Program, program number 5740-XY6) and a 3848 Cryptographic Unit,

data enciphering operations are performed with the CIPHER macro instruction.

The OS/VS CIPHER macro instruction enciphers and deciphers data and is the functional equivalent of the 4700 ENCIPHER and DECIPHER instructions.

You can use the CIPHER macro to encipher plaintext being sent to a 4700 controller, or to decipher ciphertext being received from a 4700 controller. The CIPHER macro can also be used to emulate the 4700 MACGEN instruction (see Chapter 5, "Authenticating Messages").

The OS/VS cryptographic instructions, including CIPHER, are described in *OS/VS1 OS/VS2 MVS Cryptographic Unit Support: Installation Reference Manual,* SC28-1016, and *OS/VS1 OS/VS2 MVS Programmed Cryptographic Facility: Installation Reference Manual,* SC28-0956.

Even though a 4701 application program and a host application program encipher and decipher data with different instructions, the two programs can still interact and exchange enciphered data. However, to ensure that information is usable once deciphered, the two programs must reach agreement on a set of rules or protocols to govern the manner in which the data is exchanged.

For example, the two programs must determine who selects the key and ICV and how they are distributed. A key and an ICV may be selected by one program and securely transmitted to the other. Alternatively, one program may choose the key while the other chooses the ICV. To confirm delivery and installation of the key and ICV and to validate the legitimacy of the correspondent program (within some acceptable limit), the two programs should exchange an enciphered test message. During long communication sessions it may be desirable to change the data-encrypting key--perhaps several times--on command by one of the cooperating programs.

Before the two programs can exchange enciphered data, they must agree on a common method of enciphering short blocks of data (blocks not a multiple of eight bytes in length). Typically, short blocks are padded to the next multiple of eight bytes.

Note that the 4700 ENCIPHER instruction provides a parameter which, if selected, causes data to be automatically padded before encryption. The number of added bytes (1-8) are indicated in the last byte, known as the count byte. Likewise, the 4700 DECIPHER instruction can be tailored (via an appropriate parameter) to automatically remove pad bytes from a message after decryption (by reducing the data length by the amount indicated in the count byte).

The CIPHER macro does not provide automatic padding nor is there a dynamic method by which data received can be identified as having been padded. A procedure for adding and removing pad bytes to and from messages must be agreed to during the design phase of your application program.

One simple approach is to always pad before enciphering and always remove pad bytes after deciphering. The number of bytes to discard depends on the value found in the count byte. A count greater than eight should be declared an error.

A typical macro sequence needed by an ACF/VTAM program designed to exchange ciphertext with a 4701 might take the form illustrated below. To simplify the example, messages are assumed to be a fixed length of 64 bytes and therefore do not require any length adjustment (padding). It is further assumed that a key and ICV have previously been exchanged and validated.

This sample program is intentionally incomplete. It is not intended to represent executable code but rather to suggest the proper sequence of events necessary in the receipt, manipulation, and transmission of enciphered data.

```
* * * * * * * * CIPHER Macro - Transmittal of Enciphered Data * * * *
           •
           •
           •
          RECEIVE ...,RPL=RPL1,....  program obtains enciphered
                                     message from 4701; data
                                     placed in OUTAREA
           •
           •
           •
          CIPHER  CPHRTXT=OUTAREA, . The ciphertext in OUTAREA,
                  LENGTH=COUNT, .... whose length is COUNT,
                  FNC=DECPHR, ...... is deciphered
                  KEY=KEYAREA, ..... using the key in KEYAREA and
                  ICV=ICVAREA, ..... the ICV in ICVAREA
                  CLERTXT=INAREA, .. with the plaintext being placed
                  MF=S               in INAREA.
           •
           •      received message interrogated; action taken;
           •      response message generated and placed in INAREA
           •
          CIPHER  CLERTXT=INAREA ... The plaintext in INAREA,
                  LENGTH=COUNT, .... whose length is COUNT,
                  FNC=ENCPHR, ...... is enciphered
                  KEY=KEYAREA, ..... using the key in KEYAREA
                  ICV=ICVAREA, ..... and the ICV in ICVAREA
                  CPHRTEXT=OUTAREA,  with the resulting ciphertext
                  MF=S               being placed in OUTAREA.
           •
           •
           •
          SEND    ...,RPL=RPL1,....  enciphered response sent to 4701


RPL1      RPL     ...,AM=VTAM,
                  AREA=OUTAREA,....
          DS      0F ............... align on fullword
INAREA    DS      CL64 ............. plaintext message
OUTAREA   DS      CL64 ............. ciphertext message
KEYAREA   DS      XL8 .............. data-encrypting key enciphered
                                     under host master key
ICVAREA   DS      XL8 .............. initial chaining value (ICV)
COUNT     DC      C'64' ............ standard message size, 64 bytes
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Selection, Distribution, and Installation of Keys

Because the ultimate success of a cryptographic application depends upon the secrecy afforded its cryptographic keys, the process of key selection, distribution, and installation requires careful scrutiny. A high degree of security is realized when these processes are automated. The cryptographic services of a S/370 in concert with those of the 4700 system enable user-written application programs to automate the tasks of key generation, distribution, and installation in an efficient and secure manner.

At the 4701, automatic key generation is performed by the KEYGEN instruction, as previously described. The GENKEY macro instruction provides a similar service at a S/370. The 4700 RFMK instruction and the S/370 GENKEY macro instruction provide the means to encipher a key so that it can be securely transmitted to another node via the network. The 4700 RTMK instruction and the S/370 RETKEY macro instruction provide the means to install a key when it is received from another node by reenciphering it to the form needed to encipher and decipher data at the receiving node. Throughout the entire process of key generation, distribution, and installation, the subject key remains enciphered, thereby ensuring the security of subsequent data ciphering operations.

It is the task of the cooperating application programs to agree on the manner in which a key is exchanged. A series of messages is normally needed to exchange the key, the associated ICV, and to confirm delivery and correct reception. One method was suggested above in Example 3 (see "Example 3. A Key Validation Protocol" on page 4-6). Another example is suggested by the protocol specified in IBM's Systems Network Architecture (SNA) to exchange and confirm delivery of a cryptographic key (or session key) and ICV for a communications session (see *IBM Cryptographic Subsystem Concepts and Facilities*).

The philosophy of the SNA approach is that one location (node) assumes a primary status for the purpose of generating and distributing the key. The secondary location is responsible to challenge the primary location's credibility insofar as it generates the ICV and controls the handshake protocol designed to prove that the primary location is a valid session partner. This protocol makes it easy for the program to detect the replay of a previously-recorded communication session. A program using this protocol can also avoid establishing a session with a "wrong" location -- that is, with a location that has a malfunctioning cryptographic subsystem, is ineligible to support an encrypted session, or has simply been contacted by mistake.

This "handshake" protocol requires the session partner to prove its "identity" by correctly manipulating a time-variant value (such as the ICV) via a cryptographic operation based on the session key. One location can conclude that the other is a legitimate location insofar as it has demonstrated its ability to correctly manipulate the time-variant ICV via a cryptographic operation. The success of the operation demonstrates that each location's cryptographic algorithms are operating correctly. Finally, assuming the session key is originally transmitted enciphered under a key-encrypting key known by the secondary location, it demonstrates that the message which originally transmitted the session key did not arrive as the result of a routing error. This is because the session key can be recovered only by deciphering it with the specific key-encrypting used for the original encryption.

Generating Keys with the OS/VS Cryptographic Subsystem

The Programmed Cryptographic Facility (PCF) and the Cryptographic Unit Support Program (CUSP) contain key generation capabilities that can be used in place of, or to complement, those provided by the 4700 Finance Communication System. These facilities are discussed in detail in the OS/VS Cryptographic Subsystem installation reference manuals (see the Bibliography).

You can generate master keys at the System/370 host site using the key generator utility program provided by PCF and CUSP. With this program the master keys for an installation's entire set of 4700 controllers can be generated at one time and then distributed, under physically secure conditions, to the respective 4701 sites for installation. Likewise, generation and distribution of cross-domain keys may be performed at the central host site, in conjunction with master key generation and distribution.

When a cross-domain key is installed (as described above), one at a host and the other at a remote 4701, keys can be subsequently distributed via the communications network without fear of key exposure. A session key, for example, can be generated at either the host or a 4701 and securely transmitted to the correspondent location enciphered under the cross-domain key designated for such transmissions. While passing through the network the session key is protected from disclosure by the cross-domain key.

To install a host master key on a system supported by PCF, pass the plaintext master key as a parameter to PCF's key generator utility program. Note that PCF, being a software product, retains the host master key on the Cryptographic Key Data Set (CKDS), a system resource provided for the exclusive use of the System/370 key management routines. Conversely, with the 3848 Cryptographic Unit, the host master key is installed directly in the hardware via a hand-held key-pad device. A copy of the plaintext host master key is passed as a parameter to CUSP's key generator utility program wherein it is used to verify that the host master key installed in the 3848 is identical to the one used to create the on-line CKDS.

Centralized management of key generation and distribution benefits from the physical security afforded the host site and tends to simplify recovery operations following any loss at remote sites. On the other hand, decentralized control provides the opportunity for the 4700 to perform key generation and distribution independent of the host computer.

Enciphering Under Controller Variants with the OS/VS Cryptographic Subsystem

1. Use the EMK (encipher under master key) macro instruction with the plaintext variants of the controller master key (KM1, KM2, and KM3) to create eKMH(KM1), eKMH(KM2), and eKMH(KM3).

2. Use the CIPHER (encipher) macro instruction, specifying a result of Step 1 as the *key* input, and the plaintext being enciphered as the *data* input.

You should execute these instructions in a secure environment because plaintext keys are being used.

Generating Data-Encrypting Keys in an OS/VS Cryptographic Subsystem

OS/VS GENKEY and RETKEY macro instructions together provide key management functions equivalent to those provided by the 4700 RFMK and RTMK instructions.

When invoked with the GENERATE option, the GENKEY macro creates a 64-bit random number by repeatedly executing the data encryption algorithm.

The randomness of the number is achieved by driving the data encryption algorithm with variable data obtained from storage which is recognized as being highly volatile and therefore extremely unpredictable. Also used are multiple readings of the system clock taken at irregular intervals and residual random data saved from a prior execution of the generation process. To ensure the secrecy of the derived data-encrypting key, the result of the several iterations of the algorithm is defined as the data-encrypting key enciphered under the host master key. The point here is that even during the generation process, the program can't access the data-encrypting key.

The syntax of the GENKEY macro with the GENERATE option is described in the OS/VS publications.

The GENKEY macro has the advantage that a single call can produce multiple enciphered copies of a generated data-encrypting key thus simplifying subsequent key management operations. For example, should a host application program determine that cryptography is to be employed on a communications session with a designated 4701, the GENKEY macro can be invoked to obtain two copies of the generated data-encrypting key; the first copy, enciphered under the host master key, can be used directly with the host's CIPHER macro instruction. The second copy, enciphered under a specified cross-domain key shared with the designated 4701, allows the enciphered data-encrypting key to be sent securely to the 4701. Recovery at the 4701 is accomplished with the 4700 RTMK instruction.

The OS/VS Cryptographic Subsystem Key Data Set

The cryptographic key data set (CKDS) is the repository for key-encrypting keys at a host S/370. The CKDS is an exclusive resource of the host key management routines embodied in the key generator utility program and the programs supporting the GENKEY and RETKEY macro instructions. Collectively, these programs are referred to as the host's key manager.

The key generator utility program is used to create and maintain the CKDS. Accordingly, it has read/write access privileges to the CKDS. The routines that support the GENKEY and RETKEY macro instructions have read access privileges to the CKDS so they can obtain the host's copy of the various key-encrypting keys used to satisfy GENKEY and RETKEY requests. Access to the CKDS is denied to all other programs.

Keys are enciphered under one of two variants of the host master key before being written to the CKDS. With the exception of the host master key for a system supported by PCF, plaintext keys are never stored on the CKDS. Note that a S/370 host does not define a master key variant exclusively for PIN management as does a 4701.

An Example of Generating Data-Encrypting Keys in OS/VS

The following example illustrates the coding of the GENKEY macro by a S/370 VTAM application program. The example assumes the designated cross-domain key has previously been installed on the CKDS and is associated with the label K4701001.

This example is also applicable for the generation and exchange of a key to be used for message authentication. The GENKEY macro instruction can also be used to generate random 64-bit values which can subsequently be defined (after parity is adjusted) as keys for private cryptographic applications -- much like the 4701 KEYGEN instruction. The 64-bit value generated by GENKEY (or KEYGEN) can also be used as an initial chaining value.

```
* * * * * * GENKEY Macro - Data-Encrypting Key Generation * * * * * * *
            •
            •
            GENKEY  GENERATE, ........ generate data-encrypting key:
                    OPKEY=KEYAREA, ... enciphered under host master key
                    LOCKEY=ANSAREA ... enciphered under cross-domain
                                       key
            •
            •
            LA      1,MESSAGE ........ addressability to MESSAGE
            LA      2,KEYVALUE ....... addressability to data-
                                       encrypting key enciphered
                                       under designated cross-
                                       domain key
            MVC     KEYFIELD(8,1),0(2) copy enciphered key into message
            •
            •
            SEND    ...,RPL=RPL1,....  enciphered key sent to 4701
            •
            •
            CIPHER  ...,KEY=KEYAREA,.. data ciphering with generated
                                       data-encrypting key
            •
            •
RPL1        RPL     ...,AM=VTAM,
                    AREA=MESSAGE,...
            DS      0F  .............. align on fullword
ANSAREA     EQU         .............. symbolic name of 16-byte area
KEYNAME     DC      C'K4701001' ...... cross-domain key label
KEYVALUE    DS      XL8 .............. data-encrypting key enciphered
                                       under cross-domain key
KEYAREA     DS      XL8 .............. data-encrypting key enciphered
                                       under host master key
MESSAGE     DSECT   .................. skeleton of message for 4701
FIELD1      DS
FIELD2      DS
   •        •
   •        •
   •        •
KEYFIELD    DS      XL8 .............. enciphered key for 4701
   •        •
   •        •
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Message Encryption Protocols

Having accomplished the task of generating, distributing, and installing a data-encrypting key (generically referred to as a session key when discussing communication security), the question remains of how often to use it and when used, for how much data. This decision can best be made by the cooperating programs, which can monitor the nature of the data being exchanged.

It has already been suggested that a decision to "encrypt everything," while it certainly provides high security, may result in severe performance degradation in a system processing a very high volume of data. This is an unavoidable by-product of cryptography--enciphering and deciphering takes time. Therefore, to balance security and performance, it is necessary to carefully examine the nature of the data being managed by a particular application.

The IBM 3624 Consumer Transaction Facility serves as an example of a cryptographic application designed to achieve data privacy--but only for selected, sensitive fields within messages: only the PIN field is enciphered, to protect the identity of the PIN.

A second encryption is performed which involves the sequence number field and seven bytes of the PIN field which amounts to a double encryption of the majority of the PIN. It is significant to note that the second encryption is performed with a key different from that used to encipher the PIN. In this manner, even though a message is deciphered (for example, at a 4701), the plaintext PIN is not revealed. (Details of 3624 encryption may be found in *3624 Programmer's Reference.)*

Using the 3624 as an example may over-simplify the answer to the question of "how much to encrypt" because the 3624 deals with limited sensitive information (that is, the PIN).

With an application that must manage a wide variety of message types, it may be wise to encipher some messages in their entirety, encipher parts of others (sensitive fields, for example), and avoid encrypting altogether for the remainder (on the grounds that there is no information worth protecting). The nature of the decision rests with the purpose of a particular message and the nature of the data normally transmitted therein. Likewise, there must be a convenient method available to differentiate among message types thereby indicating their relative security classifications. When a particular message type is identified, a fixed set of encryption rules is enforced.

Reenciphering Keys with the OS/VS Cryptographic Subsystem

You can use the OS/VS GENKEY macro instruction with the SUPPLIED parameter to reencipher a key from encipherment under the host master key to encipherment under a specified key-encrypting key (typically to prepare a previously generated key for transmission to a remote 4700 controller). You can use the OS/VS RETKEY macro instruction to receive an enciphered key sent from a remote location. GENKEY (with the SUPPLIED parameter) and RETKEY provide functions equivalent to the 4701's RFMK and RTMK instructions, respectively, and constitute the basis for S/370 key management.

Using the GENKEY macro instruction with the SUPPLIED parameter has the advantage that key-encrypting keys (such as cross-domain keys) can be securely transferred between a host S/370 and a 4701 with no dependency on the master keys installed at each location. Rather, the initial pair of manually installed cross-domain keys shared between a host and a 4701 are used to encipher other key-encrypting keys which are subsequently transferred between the two locations via the communication network. You can distribute keys rapidly in this manner, without relying on manual key delivery methods. Security is also enhanced insofar as there is a marked reduction in the dependency on human intervention because the process is automated and under the control of the host computer and the 4701.

The syntax of the GENKEY macro instruction with the SUPPLIED parameter is described in the OS/VS Cryptographic Subsystem publications.

Typically, key-encrypting keys, being long-term keys, are generated in plaintext form and manually installed in their intended cryptographic devices. Automated key distribution is possible provided these plaintext keys can first be enciphered making them eligible to participate with the GENKEY macro instruction.

The Encipher Under Master Key (EMK) macro instruction can be used to encipher a plaintext key under the host master key. A data-encrypting key enciphered under the host master key is in the correct form to be used with the CIPHER macro instruction or with the GENKEY macro instruction (with the

SUPPLIED parameter). A key-encrypting key enciphered under the host master key is in the correct form to be used with the GENKEY macro instruction (with the SUPPLIED parameter). It is important to note that there is no inverse function for EMK; the cryptographic subsystem provides no means to decipher an enciphered key such that a plaintext key is returned to the program.

The syntax of the EMK macro instruction is illustrated in the OS/VS Cryptographic Subsystem publications.

To illustrate the use of the GENKEY macro instruction with the SUPPLIED parameter, consider the task of an installation security specialist who must arrange for the exchange of a key between a host S/370 and a 4701 to be used to authenticate messages transferred between the two locations (that is, a message authentication key, KMAC). And, for this example, KMAC is assumed originally generated at the S/370 in plaintext. Therefore, before invoking the GENKEY macro instruction, KMAC must first be enciphered under the master key of the host. This is a reasonably straightforward task using the EMK macro. The result of the EMK macro instruction would be written as eKM0(KMAC), and read as "KMAC enciphered under the host master key."

Several things have been accomplished at this point. Due to the fact that KMAC has been enciphered, its identity is protected from disclosure. KMAC is also in the correct form for use with the GENKEY macro instruction insofar as it may be reenciphered under a cross-domain key and securely transmitted to one or more 4700 controllers. In its enciphered form, KMAC may be used with the CIPHER macro instruction to generate message authentication codes (MACs) discussed in Chapter 5, "Authenticating Messages."

As a result of executing the GENKEY macro (specifying SUPPLIED) with eKM0(KMAC) as the OPKEY parameter, and identifying a cross-domain key (represented as KCD1 and assumed shared between the host and a designated 4701) via the LOCKEY parameter, the GENKEY macro will produce eKCD1(KMAC) which may be transmitted directly to the designated 4701. Recovery of KMAC in a usable form at the 4701 is accomplished via the 4700 RTMK (as described above) instruction using a copy of KCD1 stored at the 4701.

Although the discussion thus far suggests that key exchanges should be initiated by a host S/370, such action can also be initiated by a remote 4701. Therefore, the host S/370 must have the capacity to receive an enciphered key from a 4701 and be able to convert it from the form used for transmission to the form required for use with the host's CIPHER macro instruction.

The RETKEY macro instruction is functionally equivalent to the 4700 RTMK instruction and provides the host S/370 with the ability to transform a key enciphered under a key-encrypting key (such as a cross-domain key) to encipherment under the host master key.

The syntax of the RETKEY instruction is described in the OS/VS Cryptographic Subsystem publications.

As an additional security option, PCF and CUSP enable an installation to define selected key-encrypting keys for use by only privileged programs. During execution, the RETKEY macro instruction interrogates the status of the calling program and the characteristics of the designated key as recorded on the CKDS (discussed below), and based upon this information, ends if a non-privileged program requests the use of a restricted key-encrypting key. The OS/VS Cryptographic Subsystem publications describe this feature in their respective treatment of the key generator utility program.

The following example illustrates the coding of the RETKEY macro instruction by a S/370 ACF/VTAM program responsible for receiving a data-encrypting key from a remote 4701 and transforming it for use with the CIPHER macro instruction. This example assumes the designated cross-domain key has previously been installed on the CKDS and is associated with the label K4701001.

```
* * * * * * * * * RETKEY Macro - Received Key Recovery * * * * * * * * *
             •
             •
             •
           RECEIVE  ...,RPL=RPL1,....  program obtains message from
                                       4701; data placed in INAREA
             •
             •
             •
           LA       2,MESSAGE ........ addressability to MESSAGE
           LA       1,INKEY .......... addressability to KEYFIELD in
                                       work area
           MVC      0(8,1),KEYFIELD(2) copy enciphered data-encrypting
                                       key into work area
           RETKEY   OPKEY=OUTKEY, .... transform data-encrypting key
                    REMKEY=KEYNAME ... from encipherment under the
                                       cross-domain key to under the
                                       host master key
             •
             •
             •
           CIPHER   ...,KEY=OUTKEY,... data ciphering using trans-
                                       formed data-encrypting key
             •
             •
             •
RPL1       RPL      ...,AM=VTAM,
                    AREA=INAREA,....
            DS      0F  .............. align on fullword
KEYNAME     DC      X'K4701001' ...... key-encrypting key name
INKEY       DS      XL8 .............. data-encrypting key enciphered
                                       under key-encrypting key
OUTKEY      DS      XL8 .............. data-encrypting key enciphered
                                       under host master key
MESSAGE     DSECT   .................. received message
FIELD1      DS
FIELD2      DS
             •
             •
             •
KEYFIELD    DS      XL8 .............. location of enciphered data-en-
                                       crypting key in received message
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Message Authentication Based on KMAC

You use the MACGEN instruction to generate a MAC at a 4700 Finance Communication System. To generate a MAC at the host S/370 with PCF or CUSP installed, you use the CIPHER macro instruction and additional user programming to locate and retrieve the MAC in the generated ciphertext.

Message authentication and message encryption can be applied independently. That is, a message can be authenticated regardless of whether the message is transmitted entirely in plaintext, partly in ciphertext (field encryption), or entirely in ciphertext (message encryption).

The purpose of message authentication is to detect message alteration and message substitution, typically the result of an active attack against data in transit. Message authentication is effective when the method is sufficiently complex to discourage even the most determined adversary. A complex algorithm need not be designed specifically to satisfy the requirements of authentication. A recognized strong cryptographic algorithm suffices for message authentication applications. Additional security is attained when the secret key used to personalize the algorithm (KMAC) is randomly chosen from a very large set of possible keys, is different from one communication session to another, and is routinely changed. (Under some circumstances KMAC and KS, the session key, may be one in the same with no loss in security.)

When message authentication and message encryption (for entire messages or fields within messages) are used together, you must determine when each event is to take place. When a single key is used for data encryption and authentication, the MAC should be generated on the ciphertext. If different keys are used, you can generate the MAC on either the ciphertext or the plaintext. Regardless, it is necessary to coordinate the input to the message authentication algorithm by both message sender and message receiver to ensure their ability to generate equivalent message authentication codes (MACs).

Message authentication provides protection against message alteration and message substitution but does not immediately identify the replay of a prior, valid message (assuming the authentication key and/or ICV has not changed). Replay detection, which normally must be performed by the application program, is a relatively simple task when messages contain a nonrepeating sequence number or timestamp as part of the data. Following a successful MAC comparison, the receiving application program must examine the sequence number or timestamp (after deciphering the message if appropriate) to determine if the message is current. Current messages are processed while messages that are "stale" (not current) are ignored. It is not unusual to also record the event by making an entry in an appropriate journal or log.

A method of selecting, distributing, and installing a key to control the message authentication process (KMAC) can be very similar to that described earlier for session keys.

Note: KMAC should not be distributed enciphered under KS if the identity of KMAC is to be kept secret. Rather, KMAC should be distributed after it is enciphered under an appropriate cross-domain key, as was KS, and recovered with either the 4700 RTMK instruction or the OS/VS RETKEY macro. The result of either of these instructions is KMAC enciphered under the master key of the respective location; this is the correct form for later MAC generation operations (4700 MACGEN instruction or OS/VS CIPHER macro). If KMAC is transmitted doubly enciphered under KS -- that is, eKS(eKCDs(KMAC)), verifying KMAC is implied after successfully deciphering with KS. The following diagram illustrates KMAC generation, distribution, and recovery between 4700s and S/370s.

```
* * * * * * KMAC Generation, Distribution, and Recovery * * * * * * * *

          Generation Point                   Recovery Point
          ================                   ==============
4700:
-----
KEYGEN: ===> RN = eKCD2(KMAC)

  RTMK: eKM2(KCD2), eKCD2(KMAC)
        ===> eKM(KMAC)
                                          4700:
                                          -----
  RFMK: eKM1(KCD1), eKM(KMAC)
        ===> eKCD1(KMAC) ---------------> RTMK: eKM2(KCD1), eKCD1(KMAC)
                                |               ===> eKM(KMAC)
                                |
                                |         S/370:
                                |         ------
S/370:                          |-------> RETKEY: eKM2(KCD1), eKCD1(KMAC)
------                                            ===> eKM0(KMAC)
GENKEY: GENERATE, eKM1(KCD1)
        ===> eKM0(KMAC), eKCD1(KMAC)
                                |         4700:
                                |         -----
                                |
                                |-------> RTMK: eKM2(KCD1), eKCD1(KMAC)
                                |               ===> eKM(KMAC)
                                |
                                |         S/370:
                                |         ------
                                |-------> RETKEY: eKM2(KCD1), eKCD1(KMAC)
                                                  ===> eKM0(KMAC)
Legend:
     RN = Random number
    KM0 = Host master key
     KM = Controller master key
    KM1 = Master key variant 1 of either S/370 or 4700
    KM2 = Master key variant 2 of either S/370 or 4700
   KCD1 = Sending cross-domain key
   KCD2 = Receiving cross-domain key
   KMAC = MAC generating key
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

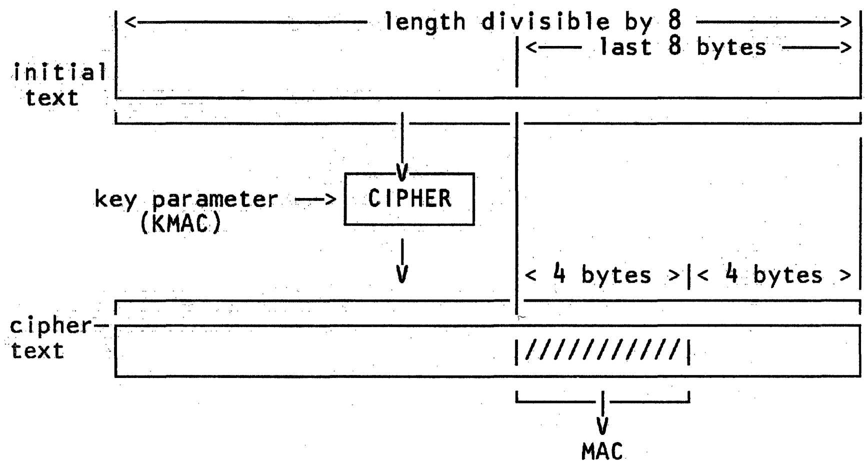
Emulating the 4700 MACGEN Instruction with the OS/VS Cryptographic Subsystem

A message authentication code (MAC) can be generated for any number of contiguous data bytes at a S/370 using the CIPHER macro instruction. However, unlike the 4700's MACGEN instruction, which directly yields a generated MAC in an area identified by one of the instruction's parameters (that is, the one used to initially locate the ICV), the caller of the CIPHER macro instruction must manually identify the MAC at a prescribed location in the resulting ciphertext.

Note that the authenticated data is actually enciphered under KMAC. If you do not want the data enciphered under KMAC, copy the data to a work area prior to invoking CIPHER.

To produce a MAC compatible with that created by the 4700 MACGEN instruction, the data item to be authenticated must first be padded (if necessary) and enciphered with the same data-encrypting key, ICV, and pad character used by the cooperating 4701.

The first 4 bytes of the last 8-byte block of ciphertext represents the MAC. The following figure illustrates the location and size of a MAC generated for a data item assumed to be longer than eight bytes.



The host program containing the CIPHER macro, used to generate a MAC, requires additional instructions designed to locate that portion of the ciphertext that represents the MAC. This code would likely identify the end of the storage containing the ciphertext by adding the value addressed by the LENGTH parameter to the starting address of the ciphertext specified by the CPHRTXT parameter. When the end of the ciphertext has been determined, the program then can subtract eight (bytes) from this address to determine the beginning address of the MAC.

Typically, data transmitted in plaintext is subject to alteration and replacement while in transit. Using a MAC with data transmitted in plaintext offers a high degree of protection from active attacks intent on altering or replacing transmitted data. Enciphered data is not precluded from active attacks intent on replacing the data with previously transmitted ciphertext. Unless the key and/or ICV used to generate a MAC is frequently changed, a MAC check cannot detect the replay of a prior, valid message. Detection of such an attack requires a variable in the message data (such as a sequence number or a time-stamp) which must be interrogated by the program after deciphering. A message found to be out of sequence or one whose time stamp is outside an acceptable range, must be considered suspect and invoke appropriate recovery action.

A program that must contend with a MAC must also be concerned with the generation, distribution, and verification of the key used to generate the MAC. For just as it is necessary for two programs to each have a copy of the same cryptographic key to exchange enciphered data, so must two programs have a copy of the same key to generate identical MACs.

The method of selection, distribution, and verification of a MAC generating key (KMAC) is essentially the same as that described for a data-encrypting key (see Chapter 3) providing that the cooperating programs declare, based upon their adopted protocol, when a data-encrypting key is being exchanged and/or when a

MAC generating key is being exchanged. Clearly, the effect of one program using one key for one purpose and the other program using a different key for the same purpose would become rapidly apparent when MAC checks continually fail and data does not decipher correctly.

When you are using both message encryption and message authentication, it is essential that the two programs establish the sequence of these operations. When message encryption precedes MAC generation (that is, a MAC is generated on ciphertext), the same key can be used for both operations with no loss in security. But if you generate the MAC *before* you encipher the data, an opponent could alter ciphertext with a high degree of confidence that the alteration will not be detected by a subsequent MAC check. Although not necessarily a practical method, using different keys for message encryption and MAC generation will also prevent this attack from being successful.

For example, compromise of a data-encrypting key results in a loss of data secrecy but does not permit successful data alteration or substitution because a MAC cannot be generated on arbitrary data without the correct KMAC. A compromise of KMAC (assuming enciphered data) allows an opponent to replay previously transmitted data (such as a "deposit" message) without detection, but does not yield the detailed content of the data being replayed (the amount of the "deposit" is not visible in ciphertext). If a single key is used for both purposes, and is compromised, all the security benefits cryptography has to offer are lost.

Host PIN Security

The most practical method of PIN validation you can perform at a host S/370 (which does not have an equivalent of the 4700 PINVERIF instruction), is to compare a received PIN against its presumed identical counterpart stored on a host PIN data base.

A host program that processes PINs in conjunction with a 4701 program must perform two functions. First, the host program must be capable of receiving a PIN from its attached communication network, and second, the host program must manage a data base of stored PINs. The identity of a PIN should be protected by encryption during all periods of storage or transmission after being entered into the system by a customer. Typically, the PIN protection key used to initially encipher the PIN at the system entry point (KP1) should not be the same as the key used to encipher the PIN stored on the PIN data base. You should use an output PIN protection key (KP2, not equal to KP1) for PIN exchanges between the 4700 and the host. Ideally, KP2 should not also be the key used to encipher the PINs residing on the host data base. This being the case, the host, therefore, must also be capable of reenciphering a PIN from one key (the transmission key, KP2) to another (the PIN data base key).

Reenciphering a PIN from one key to another is possible by a S/370 host with PCF or CUSP. With either of these program products, reenciphering a PIN is performed by executing the RETKEY macro instruction followed by the GENKEY macro instruction (with the SUPPLIED parameter). When these two macros are supplied with the correct key parameters, a user-entered PIN is reenciphered under the PIN data base key. After it is obtained, the transformed user-entered PIN is compared with the corresponding PIN stored on the PIN data base for equality. An identical comparison is required to approve the related transaction requested by the customer.

PIN Formats

Because a PIN may assume one of several different formats before being initially encrypted at an entry point into the system, it cannot be used for comparison purposes against its counterpart stored on a S/370 host PIN data base. Likewise, the key used to encipher a PIN at a system entry point (KP1) should not be the same key used to protect the PIN stored on a host PIN data base. Therefore, the format and the encipherment of a PIN provided at a system entry point must be altered before the PIN may be used for comparison against its host counterpart.

The most practical place to perform this task is at the 4700 controller. The 4700 PINTRANS instruction performs the desired reformatting, and simultaneously reenciphers the PIN from the key used at the system entry point (KP1) to a key used to send PINs to a S/370 (for example, an output PIN protection key, KP2).

However, understand that you cannot arbitrarily determine the format of the PIN. The PINTRANS instruction supports only one output PIN format. In order that meaningful comparisons are performed at a host, the PINs stored on the host PIN data base must conform to this single format. This restriction is necessary because the GENKEY and RETKEY macros (used to reencipher a PIN) do not yield a plaintext PIN, thereby denying an application program the opportunity to modify a PIN's format. Therefore, you should format the PIN at the 4700 controller.

> **Note:** Before the PINs can be formatted, enciphered, and stored on the host PIN data base, the PINs must be recorded in unformatted, plaintext form. You will have to securely maintain this record of plaintext PINs so that you can tell your customers what PIN has been assigned to them. The unformatted, plaintext PINs must be generated, recorded, and enciphered under physically secure conditions.

Managing PINs with the OS/VS Cryptographic Subsystem

Unlike host modules BDKDPRS and BQKDPRS, neither PCF nor CUSP provides explicit services intended to address the specific task of PIN generation. However, the GENKEY macro instruction, being much like that of the 4700 KEYGEN instruction, generates random 64-bit strings. You can manipulate these strings (truncate, concatenate) and redefine them as plaintext PINs.

Note though that the generation process that yields a PIN in plaintext (or any secret value for that matter) must be coupled with some immediate action designed to conceal the identity of the PIN from other programs. In some cases, depending upon the method of PIN validation used by a particular installation, the PIN is placed in some established format, enciphered, and stored on an appropriate data base. In other cases, the PIN participates in one or more transformation processes designed to establish nonsecret parameters from which the PIN may be regenerated at some future time when it is necessary to validate a PIN entered by a customer. In this latter case, the PIN is recovered through the use of a secret parameter (for example, a system managed cryptographic key) known only to the installation that originally issued the PIN.

PIN validation can subsequently be performed in one of two fundamental ways. Either a calculated PIN of reference is produced on demand and compared with the presumed identical PIN provided by an institution's customer, or a customer's PIN is enciphered and compared with a similarly enciphered PIN of reference obtained from an appropriate data base.

A PIN can be dynamically validated at a 4701 controller. For those cases where an institution determines that validation should or may be performed at a host S/370 rather than at a 4701, a PIN validation design based upon static values is chosen.

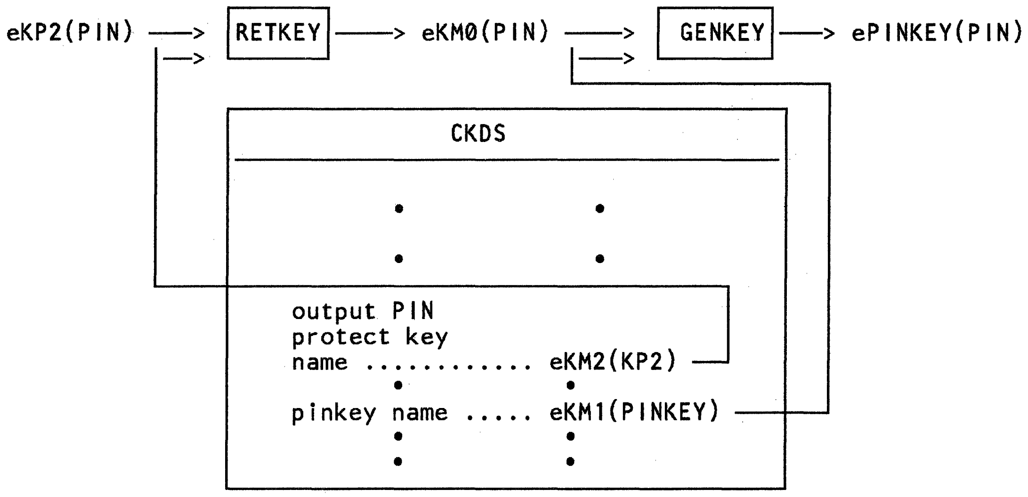
With such a design the central host would manage a data base of PINs enciphered under a special "pinkey" for all customers of the institution. PINs would be transmitted to the central host enciphered under one of several output PIN protect keys (KP2) which, in this case, operate much like a cross-domain key. After it receives the PIN, the host program could conveniently and securely transform the PIN from encryption under KP2 to encryption under the special "pinkey" by sequentially exercising the RETKEY and GENKEY macro instructions (described in the following section). The enciphered PIN stored on the host's data base would then be retrieved, via the account number which must accompany the transmitted PIN. A comparison of the two enciphered quantities for equality would be performed to determine the correspondence between the PIN entered by the customer and the stored PIN of reference.

Emulating the PINTRANS Instruction with the OS/VS Cryptographic Subsystem

The 4700 PINTRANS instruction can be emulated at a S/370 insofar as a PIN can be reenciphered from one key to another without exposing the PIN in plaintext. However, it is not possible to reformat a PIN, as can be done with the 4700 PINTRANS instruction.

You can reencipher a PIN by executing the RETKEY macro instruction and the GENKEY macro instruction in series. The key-encrypting keys designated by each of the macros must have been previously installed on the CKDS (enciphered under the correct variant of the host master key) to ensure the final result is meaningful ciphertext and not a useless, random string of bits. The key-encrypting key used to encipher the PIN during transmission from the 4701 to the S/370 (typically an output PIN protect key, KP2, functioning as a cross-domain key) must be enciphered under the second variant of the host master key and identified via the REMKEY parameter of the RETKEY macro instruction. The key-encrypting key used to encipher the PIN stored on the S/370 data base must be enciphered under the first variant of the host master key and identified by the LOCKEY parameter of the GENKEY macro instruction.

The following diagram illustrates how these two macro instructions transform the PIN. The diagram also shows the additional key parameters needed to accomplish the PIN translation. Note that the intermediate result, eKM0(PIN) (after RETKEY completion), being an enciphered PIN, maintains the secrecy of the PIN during the interval between the RETKEY and GENKEY macro instructions.



```
   KP2 = Output PIN protect key
   KMØ = Host master key
   KM1 = Host master key variant 1
   KM2 = Host master key variant 2
PINKEY = PIN encryption key
   PIN = PIN to be verified
```

The data items addressed by the various parameters of the RETKEY and GENKEY macro instructions (input and output) are as follows:

```
RETKEY    OPKEY  = eKMØ(PIN),
          REMKEY = (input) CKDS name of eKM2(KP2)
                   (input) eKP2(PIN)
GENKEY    SUPPLIED,
          OPKEY  = (input) eKMØ(PIN),
          LOCKEY = (input) CKDS name of eKM1(PINKEY)
                   (output) ePINKEY(PIN)
```

Note that what has previously been described as key management functions are also quite effective for the secure management of PINs. And using the key management functions in this manner does not decrease the security afforded keys or PINs. This example also demonstrates effective PIN management at a S/370 host using only the two defined variants of the host master key.

Appendix A. Machine Instruction Formats

The *macro instruction formats* described in Chapter 10, "4700 Cryptographic Instructions" are provided by macro definitions within the 4700 Host Support program. The *instruction functions* described throughout this book are provided by the 4700 controller and its IBM-provided modules of controller data (P28 and P57).

This appendix provides the *machine instruction formats* for the encryption instructions. Use these formats if you wish to invoke the instruction function but do not have access to the Host Support program's macro definitions. You may also find this information useful for debugging.

The basic machine instruction format includes an operation code which may be followed by one or two addresses. All register and segment references are binary, and in the range of 0 to 15.

Note that the instructions that point to a single parameter list can have any of three formats, depending on the type of pointer passed to the macro definition. The instructions that point to two parameter lists (PINTRANS and PINVERIF) can have any of nine formats.

DECIPHER

```
 ---------------------------------------------------------
|  7B  |  68   |  00   | S2 | 0  |           D2           |
 ---------------------------------------------------------
0      8       16      24   28   32                       48
```

```
 ------------------------------------
|  7B  |  78   |  00   | R2 | 0  |
 ------------------------------------
0      8       16      24   28   32
```

```
 ---------------------------------------------------------
|  7B  |  88   |  00   | R2 | 0  |          D2            |
 ---------------------------------------------------------
0      8       16      24   28   32                       48
```

DECODE

```
+-----------------------------------+
|  57   |  S1    |  S2    |0001| D2 |
+-----------------------------------+
0       8       12       16   20   32
+---------------------------------------------------+
|  7C   |  33    |  0     | S1 | S2 | 0 |  D2       |
+---------------------------------------------------+
0       8       16       20   24   28  32          48
+-----------------------------------------+
|  7C   |  43    |  0     | S1 | R2 | 0   |
+-----------------------------------------+
0       8       16       20   24   28     32
+---------------------------------------------------+
|  7C   |  53    |  0     | S1 | R2 | 0   | D2      |
+---------------------------------------------------+
0       8       16       20   24   28     32       48
```

ENCIPHER

```
+---------------------------------------------------+
|  7B   |  65    |  00    | S2 | 0  |       D2      |
+---------------------------------------------------+
0       8       16       24   28   32              48

+-----------------------------------+
|  7B   |  75    |  00    | R2 | 0  |
+-----------------------------------+
0       8       16       24   28   32

+---------------------------------------------------+
|  7B   |  85    |  00    | R2 | 0  |     D2        |
+---------------------------------------------------+
0       8       16       24   28   32              48
```

ENCODE

```
+-----------------------------------+
|  57   |  S1    |  S2    |0000| D2 |
+-----------------------------------+
0       8       12       16   20   32
+---------------------------------------------------+
|  7C   |  35    |  0     | S1 | S2 | 0   |  D2     |
+---------------------------------------------------+
0       8       16       20   24   28     32       48
+-----------------------------------------+
|  7C   |  45    |  0     | S1 | R2 | 0   |
+-----------------------------------------+
0       8       16       20   24   28     32
+---------------------------------------------------+
|  7C   |  55    |  0     | S1 | R2 | 0 |   D2      |
+---------------------------------------------------+
0       8       16       20   24   28  32          48
```

KEYGEN

```
  7B  |  6D  |  00  | S2 | 0 |      D2
0     8      16     24   28  32            48
```

```
  7B  |  7D  |  00  | R2 | 0
0     8      16     24   28  32
```

```
  7B  |  8D  |  00  | R2 | 0 |     D2
0     8      16     24   28  32            48
```

MACGEN

```
  7B  |  6A  |  00  | S2 | 0 |      D2
0     8      16     24   28  32            48
```

```
  7B  |  7A  |  00  | R2 | 0
0     8      16     24   28  32
```

```
  7B  |  8A  |  00  | R2 | 0 |     D2
0     8      16     24   28  32            48
```

PINTRANS

```
 ______________________________________________________________
|  7F   |  03   |  00   | S2 | 0  |      D1      |      D2      |
0       8       16      24   28   32             48             64

 ______________________________________________
|  7F   |  23   |  00   | R2 | S1 |      D1      |
0       8       16      24   28   32             48

 ______________________________________________________________
|  7F   |  13   |  00   | R2 | S1 |      D1      |      D2      |
0       8       16      24   28   32             48             64

 ______________________________________________
|  7F   |  63   |  00   | S2 | R1 |      D2      |
0       8       16      24   28   32             48

 ________________________________
|  7F   |  83   |  00   | R2 | R1 |
0       8       16      24   28   32

 ______________________________________________
|  7F   |  73   |  00   | R2 | R1 |      D2      |
0       8       16      24   28   32             48

 ______________________________________________________________
|  7F   |  33   |  00   | S2 | R1 |      D1      |      D2      |
0       8       16      24   28   32             48             64

 ______________________________________________
|  7F   |  53   |  00   | R2 | R1 |      D1      |
0       8       16      24   28   32             48

 ______________________________________________________________
|  7F   |  43   |  00   | R2 | R1 |      D1      |      D2      |
0       8       16      24   28   32             48             64
```

PINVERIF

```
 7F  | 02  | 00  | S2 | 0  |    D1    |    D2
0     8    16    24   28   32        48        64

 7F  | 22  | 00  | R2 | S1 |   D1
0     8    16    24   28   32       48

 7F  | 12  | 00  | R2 | S1 |    D1    |    D2
0     8    16    24   28   32        48        64

 7F  | 62  | 00  | S2 | R1 |    D2
0     8    16    24   28   32       48

 7F  | 82  | 00  | R2 | R1
0     8    16    24   28   32

 7F  | 72  | 00  | R2 | R1 |    D2
0     8    16    24   28   32       48

 7F  | 32  | 00  | S2 | R1 |    D1    |    D2
0     8    16    24   28   32        48        64

 7F  | 52  | 00  | R2 | R1 |    D1
0     8    16    24   28   32        48

 7F  | 42  | 00  | R2 | R1 |    D1    |    D2
0     8    16    24   28   32        48        64
```

RFMK

```
 ____________________________________________________________
|   7B   |   6B   |   00   | S2 | 0 |           D2           |
|________|________|________|____|___|________________________|
0        8        16       24   28  32                       48
```

```
 _________________________________________
|   7B   |   7B   |   00   | R2 | 0       |
|________|________|________|____|_________|
0        8        16       24   28        32
```

```
 ____________________________________________________________
|   7B   |   8B   |   00   | R2 | 0 |           D2           |
|________|________|________|____|___|________________________|
0        8        16       24   28  32                       48
```

RTMK

```
 ____________________________________________________________
|   7B   |   6C   |   00   | S2 | 0 |           D2           |
|________|________|________|____|___|________________________|
0        8        16       24   28  32                       48
```

```
 _________________________________________
|   7B   |   7C   |   00   | R2 | 0       |
|________|________|________|____|_________|
0        8        16       24   28        32
```

```
 ____________________________________________________________
|   7B   |   8C   |   00   | R2 | 0 |           D2           |
|________|________|________|____|___|________________________|
0        8        16       24   28  32                       48
```

Appendix B. COPY File Fields

This appendix shows the overlays generated by the COPY DEFxxx instruction. In these overlays, "DEFxx" means either DEFLD or DEFRF.

DEFENC

```
ENCPAR   DEFxx   s,,24          INSTRUCTION PARAMETER LIST
ENCFLG   DEFxx   s,ENCPAR,1     FLAG BYTE:
ENCPADM  EQUATE    X'80'                  PAD OPTION
ENCPADC  DEFxx   s,,1           PAD CHARACTER VALUE
ENCKEY   DEFxx   s,,8           KEY (ENCRYPTED UNDER MASTER KEY)
ENCMAC   DEFxx   s,,4           MSG AUTHENTICATION CODE (MAC)
ENCIV    DEFxx   s,ENCMAC,8     INITIALIZATION VECTOR
         DEFxx   s,,1           RESERVED
ENCSEG   DEFxx   s,,1           SEGMENT NUMBER OF DATA
ENCDISP  DEFxx   s,,2           DISPLACEMENT TO DATA
ENCLEN   DEFxx   s,,2           LENGTH OF DATA
```

DEFINP

```
INPPAR   DEFxx   s,,35          INSTRUCTION PARAMETER LIST
INPINTYP DEFxx   s,INPPAR,1     INPUT PIN FORMAT TYPE
INPCLEAR EQUATE    X'00'                  CLEAR PIN FORMAT
INPEPPAD EQUATE    X'80'                  ENCRYPTING PIN PAD PIN
                                            FORMAT
INP3624  EQUATE    X'40'                  3624 PIN FORMAT
INPANSI  EQUATE    X'20'                  ANSI X9.8 STANDARD PIN
                                            FORMAT
INPRPQ   EQUATE    X'10'                  RESERVED FOR RPQ 7B0570
INPINPAD DEFxx   s,,1           INPUT PIN PAD CHARACTER
INPINKEY DEFxx   s,,8           INPUT PIN KEY (ENCRYPTED)
INPAN    DEFxx   s,,8           INPUT PRIMARY ACCOUNT NUMBER
INPANO   DEFxx   s,INPAN,2        4 DIGITS (2 BYTE) OF ZEROES
INPANA   DEFxx   s,,6            12 DIGIT (6 BYTE) ACCOUNT NUM
INPINLEN DEFxx   s,,1           INPUT CLEAR PIN LENGTH
INPIN    DEFxx   s,,16          INPUT PIN
```

DEFKYG

```
KYGPAR   DEFxx   s,,9           INSTRUCTION PARAMETER LIST
KYGFLAG  DEFxx   s,KYGPAR,1     GENERATED KEY TYPE FLAG
KYGCLR   EQUATE    X'00'                  PLAINTEXT KEY
KYGENC   EQUATE    X'80'                  ENCIPHERED KEY
KYGBUFF  DEFxx   s,,8           BUFFER FOR GENERATED KEY
```

DEFRMK

```
RMKPAR   DEFxx   s,,18          INSTRUCTION PARAMETER LIST
RMKFLAG  DEFxx   s,RMKPAR,1     RE-ENCIPHER KEY FLAG
RMKNVM   EQUATE    X'00'                  USE NVM KEY
RMKUSER  EQUATE    X'80'                  USER-SUPPLIED KEY
         DEFxx   s,,1           RESERVED
RMKINKEY DEFxx   s,,8           INPUT KEY (TO BE RE-ENCIPHERED)
RMKXDKEY DEFxx   s,,8           USER CROSS-DOMAIN KEY
```

DEFTNP

```
TNPPAR   DEFxx   s,,20          INSTRUCTION PARAMETER LIST
TNPINFLG DEFxx   s,TNPPAR,1     TRANSLATE PIN FLAG
TNPEPPAD EQUATE    X'80'                  ENCRYPTING PIN PAD FORMAT
TNP3624  EQUATE    X'40'                  3624 PIN FORMAT
TNPANSI  EQUATE    X'20'                  ANSI PIN FORMAT
TNPRPQ   EQUATE    X'10'                  RESERVED FOR RPQ 7B0570
TNPSAMEK EQUATE    X'01'                  USE INPUT PIN KEY INDICATO
TNPINPAD DEFxx   s,,1           OUTPUT PIN KEYPAD CHARACTER
TNPINSEQ DEFxx   s,,2           OUTPUT PIN SEQ NO (RPQ 7B0570)
         DEFxx   s,TNPINSEQ,1   RESERVED
TNPSEQPP DEFxx   s,,1           OUTPUT PIN SEQUENCE (PIN PAD)
TNPINKEY DEFxx   s,,8           OUTPUT PIN KEY (ENCRYPTED)
TNPAN    DEFxx   s,,8           OUTPUT PRIMARY ACCOUNT NUMBER
TNPANO   DEFxx   s,TNPAN,2      $ DIGITS (2 BYTES) OF ZEROS
TNPANA   DEFxx   s,,6           12 DIGIT (6 BYTE) ACCOUNT NUMBER
```

DEFVER3

```
VER3PAR  DEFxx   s,,35         INSTRUCTION PARAMETER LIST
V3ALGTYP DEFxx   s,VER3PAR,1   VERIFICATION ALGORITHM TYPE
VA3624   EQUATE    X'00'                 3624 VERIFICATION
         DEFxx   s,,1          RESERVED
V3PINKEY DEFxx   s,,8          PIN VERIFICATION KEY (ENCRYPTED)
V3DECTAB DEFxx   s,,8          DECIMALIZATION TABLE
V3VALDAT DEFxx   s,,8          PADDED VALIDATION DATA
V3CHKLEN DEFxx   s,,1          PIN CHECK LENGTH
V3OFFSET DEFxx   s,,8          PIN OFFSET DATA
```

Appendix C. Troubleshooting the Cryptographic Facilities

Automatic Testing

The controller tests its cryptographic storage in two ways each time you IPL the controller.

The controller's first test determines whether the storage can have data written into it and read from it. If there is a problem with the storage, the controller displays an E00A error message on its LED indicators. An E00A error message generally means that the cryptographic storage must be replaced; follow the trouble reporting procedures described in *4701 Controller Operating Instructions.*

The controller then tests the storage in another way, by using the controller master key to encipher data (this test is performed only if the P28 module is present). If the controller encounters a problem enciphering the data, it turns on the controller alert light and returns the following log message:

```
11 HHMM 021 NVM ERROR ENCIPHER STATUS = xxxx
```

where xxxx is one of the possible SMSDST status bits that can be returned from ENCIPHER (see Appendix E, "Status Codes").

This log message generally means that the master key isn't present (cleared, or never loaded) or that the cryptographic storage battery has failed. If you receive this log message, refer to *4700 Subsystem Problem Determination Guide.*

Testing With the System Monitor

The 4700 system monitor features a *320 command* that can assist you in determining whether your cryptographic facilities are working properly.

See *4700 Subsystem Operating Procedures* for a detailed description of the 320 command.

The 320 command tests the controller's cryptographic facilities in the following manner:

- The controller first determines if its 3600-level of cryptographic support (the P57 module) has been loaded. If it is present, the controller tests the module and returns a *90092* message if a problem exists, or a *10074* message if no problem exists. If you receive a 90092 message, follow the problem reporting procedures described in *4701 Controller Operating Instructions.*

- The controller next determines if its 4700-level of cryptographic support (the P28 module) has been loaded. If it is present, the controller tests the module and returns a *90093* message if a problem exists, or a *10073* message if no problem exists. Possible reasons for a 90093 message include a failure of the cryptographic storage battery, and failure to load a master key into the controller; refer to *4700 Subsystem Problem Determination Guide.*

  You can run a key verification procedure to determine if the proper master key has been loaded. This procedure is summarized earlier in this book; also see the 330-3 command in the *4700 Subsystem Operating Procedures.*

When you have verified that the controller contains a master key, the 90093 message usually means that the controller is malfunctioning; follow the problem reporting procedures described in *4701 Controller Operating Instructions.*

If the P28 module is present, the controller makes a further test of its data encryption algorithm, and returns a *90094* message if a problem exists. This message means that repair is needed; follow the problem reporting procedures described in *4701 Controller Operating Instructions.*

Note: You should erase all cryptographic keys before submitting a 4700 controller to a local service center for repair.

Appendix D. Program Check Codes

If the 4700 controller encounters an execution request that indicates a logic error, a program check results. The following are the hexadecimal codes and the explanations for possible program checks:

Code **Explanation**

01 Invalid segment specification: An operand specifies a segment that was not defined during controller configuration procedure, or segment 14 was specified in an instruction that will cause data to be stored or changed in segment 14.

02 Segment overflow: Completion of the instruction requires more storage than the specified segment provides.

03 Field length error: An incorrect field was specified. The length is greater than 2 for an immediate operand; or a SETFPL instruction attempted to adjust the field length indicator to a negative value; or a value is specified which, when added to the PFP, would be greater than the segment length; or the field length was greater than 255 for a PAKSEG instruction.

04 Return-address stack error: An LRETURN instruction was issued, but the return-address stack was empty; or a branch instruction was issued, but the stack was full.

06 Instruction count threshold: The number of instruction executions allowed per transaction has been exceeded.

08 No overlay name: The overlay name is not in the resident overlay directory.

09 Invalid operation or segment code: The instruction operation or segment selection code specified is invalid. Make sure that any required OPTMOD coding for the instruction was entered and that any parameter fields are properly coded.

0A No entry point: There is no startup entry point specified.

0B Instruction address error: An addressing error has occurred. In the case of branch instructions, the program check address field of segment 1 will contain the address of the branch instruction.

0C Instruction count exceeded: 65,535 instructions have been executed without a release of control.

0D DEFDEL missing or incorrectly used: Either a delimiter request was made but no delimiter table was found or the table is not halfword aligned.

0E EDIT mask error: The mask used with an EDIT instruction contains an error.

0F Invalid link write control field: The link write control field or write options are invalid.

10 Communication link write length error: Data length exceeds 4095, data length during an LWRITE in batch mode was too long, command data length is incorrect; negative-response data length is incorrect, or there was a negative response to setting or testing sequence numbers.

11 Invalid parameter list, or parameter space is insufficient.

12 Indexing is not active.

20 Program check in called application program.

21 Called application program not found.

22 APCALL link stack full.

23 Recursive APCALL to an application program defined as USE=STATIC during configuration.

24 APCALL storage pool defined by MAXSTOR=was exceeded.

25 APCALL segment pool defined by MAXSEG=was exceeded.

26 APRETURN issued with no APCALL link stack entry - no calling application program.

27 Register address contains invalid segment space ID.

28 No transient pool: a transient pool was not defined for this station.

29 Transient application size error: the target transient application program will not fit in the largest transient area defined in the pool for this station.

FF System error.

Appendix E. Status Codes

The controller places a status code in SMSDST when an instruction completes with an exception condition (SMSCCD=2). The following descriptions can help you determine why the controller encountered the error.

SMSDST = X'0200' (Unit Check)

The controller malfunctioned. Follow the trouble reporting procedures described in *4701 Controller Operating Instructions.*

SMSDST = X'2001' (Data Check)

Parity warning. The key being loaded into cryptographic storage (NVM) does not conform to the 'odd' parity specification of the U.S. Dept. of Commerce/National Bureau of Standards (FIPS PUB 46).

Odd parity is not required by any of the controller hardware or by any of the controller security instructions. The key is loaded into cryptographic storage as requested (unless additional status is present).

Odd parity consists of 0-7 bits in each byte of the key having an odd number of ones. Bit 7 of each byte (the parity bit) may be set or reset to insure parity.

SMSDST = X'2002' (Data Check)

Either the controller malfunctioned or there is no master key in the controller (the operator may have cleared the key). Activate the encryption keylock and load a key as described in *4700 Subsystem Operating Procedures.* If the problem persists, follow the trouble reporting procedures described in *4701 Controller Operating Instructions.*

SMSDST = X'2004' (Data Check)

Read back check failed.

When required to write to cryptographic storage (NVM), the controller writes the data, reads it, then compares the data written with the data read. If the data does not match, this status is posted.

SMSDST = X'2008' (Data Check)

Invalid pad length. The parameter list indicates pad processing, but the controller, upon deciphering the data, found a pad count that was not in the range of one to eight. Possible causes:

- You used the wrong key.
- You used the wrong initial chaining value.
- The enciphered data was not enciphered with the proper pad processing (that is, ENCIPHER or an equivalent process was not used).

SMSDST = X'2010' (Data Check)

The input PIN failed the PIN validation test.

SMSDST = X'2020' (Data Check)

The check length (number of PIN digits being checked) was greater than the length of the input PIN. This usually means that the customer entered too few digits of the PIN. Make sure the program is setting the correct check length in the verification parameter list (VER3PAR's V3CHKLEN field). This length cannot be greater than the shortest customer-entered PIN.

SMSDST = X'2040' (Data Check)

Translate length error.

While performing a translation operation, the controller encountered an invalid PIN length. The meaning of "invalid PIN length" depends on the output PIN format:

- Encrypting PIN keypad: PIN length greater than 13
- ANSI: PIN length less than 4 or greater than 12
- 3621: PIN length greater than 12

SMSDST = X'2080' (Data Check)

The input PIN was not in a valid format. For example, the PIN length may have been set to zero, or the PIN may contain one or more nondecimal digits.

- If the input PIN is in nonencrypting PIN keypad format, the problem is probably in the program; check the manner in which the program is building the parameter lists.

- If the input PIN is in one of the enciphered formats, the problem could also be caused by a malfunction at the device where the PIN was entered, or by a lack of key validation. The KP1 provided to PINVERIF may not be the same key installed in the device. You may have loaded eKM3(KP1) into the keypad, for example, instead of KP1.

SMSDST = X'8000' (Intervention Required)

The controller received a request to clear or load cryptographic storage (NVM), but found the physical keylock inactive.

Appendix F. Statistical Counters

The 4700 controller maintains a set of statistical counters for its cryptographic facilities. An operator can see the contents of the statistical counters by logging on to the system monitor and issuing a 010 command that specifies an LSSDD value of 9006. A program can obtain the contents of the statistical counters by issuing an ERRLOG or STATS instruction that specifies a physical device address of X'9060'. The controller displays (or returns) a device type of X'06'.

The controller displays the entire set of statistical counters together. In the following descriptions, the name of the counter is the displacement of the counter value within the displayed string. For example, Counter 3 occupies the third byte of the string, and Counter 9-11 is a three-byte binary value occupying the 9th, 10th, and 11th bytes of the string.

Counter 1—Machine Check

Description: The controller increments Counter 1 each time the controller encounters a machine check status.

Probable Cause: A hardware malfunction.

Recommended Action: See *4700 Subsystem Problem Determination Guide.*

Counter 2—Intervention Required

Description: The controller increments Counter 2 each time it attempts to load or clear a cryptographic key but finds that the encryption keylock has not been activated.

Probable Cause: The operator did not activate the encryption keylock, or the controller malfunctioned.

Recommended Action: Activate the encryption keylock in accordance with your institution's procedures; if this problem recurs, see *4700 Subsystem Problem Determination Guide.*

Counter 3—Invalid Key Checksum

Description: Each time the controller places a key in cryptographic storage, it calculates a *checksum* value and stores it with the key. Each time the controller reads a key from cryptographic storage, it recalculates the value and compares it with the stored value. The controller increments Counter 3 each time it reads a key and finds that the two values do not match.

Probable Cause: This problem can be caused by:

- Not loading a master key
- Erasing the master key and not loading a new one
- A controller malfunction (dead cryptographic storage battery, for example).

Recommended Action: Ensure that the controller contains a master key (330 command); if it does, see *4700 Subsystem Problem Determination Guide.*

Counter 4—Unsuccessful Write

Description: The controller increments Counter 4 each time it attempts to write into cryptographic storage and cannot do so.

Probable Cause: Controller malfunction.

Recommended Action: See *4700 Subsystem Problem Determination Guide.*

Counter 5-8—Reserved

Counter 9-11—Attempted PIN Validation

Description: The controller increments Counter 9-11 each time a PINVERIF instruction completes successfully, up to the point where the PIN is actually validated (SMSCCD = X'01', or SMSCCD = X'02' and SMSDST = X'2010'). This is the total number of PIN validations attempted. The controller provides this counter so that an operator or user-written program can audit the number of PIN validation attempts (see Chapter 6, "Validating and Translating Personal Identification Numbers").

Counter 12-14—Unsuccessful PIN Validation

Description: The controller increments Counter 12-14 each time a PINVERIF instruction completes successfully, but results in a "PIN not valid" status (SMSCCD = X'02' and SMSDST = X'2010'). The controller provides this counter so that an operator or user-written program can audit the number of unsuccessful PIN validations (see Chapter 6, "Validating and Translating Personal Identification Numbers").

Counter 15-17—Successful PIN Translation

Description: The controller increments Counter 15-17 each time a PINTRANS instruction completes successfully -- that is, completes with a condition code of 1 (SMSCCD = X'01'). The controller provides this counter so that an operator or user-written program can audit the number of PIN translations (see Chapter 6, "Validating and Translating Personal Identification Numbers").

Counter 18-32—Reserved

Glossary

This glossary defines 4700 Finance Communication System terms and other data processing and data communication terms used in this publication. This glossary includes some terms and definitions from the *IBM Vocabulary for Data Processing, Telecommunications, and Office Systems,* GC20-1699.

ciphertext. Data that is intended to be unintelligible to all except those who legitimately possess the means to reproduce the plaintext.

configuration procedure. In the 4700 system, the process of defining the configuration of a specific 4700 system to a controller. The configuration procedures can be performed either at the host computer using configuration macro instructions, or at the controller using the Local Configuration Facility (LCF).

controller data. Modules of operating information stored in the 4700 library and used to construct load images for controllers.

cross-domain keys. In 4700 systems, a pair of cryptographic keys used by a program to encipher a session key.

cryptogram. An enciphered key.

cryptographic. Pertaining to the transformation of data to conceal its meaning.

cryptographic algorithm. A set of rules that specify the mathematical steps required to encipher and decipher data.

cryptographic communication. To transmit enciphered data between devices or programs.

cryptographic key. A 56-bit value used by the DES to select one relationship between plaintext and ciphertext out of the many possible relationships the DES provides.

cryptography. The process of enciphering and deciphering data for the purpose of keeping the data secret.

customization data. Modules of operating information stored in the 4700 library and used to construct load images for the IBM 3624 Consumer Transaction Facility.

data encryption algorithm. See *Data Encryption Standard (DES).*

Data Encryption Standard (DES). A cryptographic algorithm designed to encipher and decipher data using a 56-bit key as specified in the *Federal Information Processing Standard #46,* January 15, 1977.

decipher. The process of converting ciphertext into plaintext.

DES. Data Encryption Standard.

encipher. The process of converting plaintext into ciphertext.

dummy control section (DSECT). A control section that an assembler can use to format an area of storage without producing any object code.

Host Support. A licensed program that executes in the host computer to provide services for 4700 controllers.

initial chaining value (ICV). In 4700 systems, an eight-byte pseudo-random number used for cipher block chaining.

installation diskette. An IBM-supplied diskette that enables the 4700 controller to perform a variety of installation-related procedures – for example, creating an operating diskette.

keypad. A small keyboard designed for one-handed use. For example, a PIN keypad is a small, limited-function keyboard with which an operator uses one hand to enter a personal identification number.

load image. A combination of formatted controller data, configuration data, and application program data that defines an operating environment for a 4700 controller. Load images for the IBM 3624 Consumer Transaction Facility contain customization data rather than configuration data, and include no application program data.

master key. A key-encrypting key used to encipher operational keys.

master key variant. A key-encrypting key that is derived from the master key and used to encipher other cryptographic keys.

operating diskette. The customer-generated diskette containing the load image relating to the operation of a particular controller.

operational key. A data-encrypting key used to encipher and decipher data.

parity bit. A binary digit appended to a group of binary digits to make the sum of all the digits either always odd (odd parity) or always even (even parity).

plaintext. The intelligible form of ciphertext.

plaintext keys. Unenciphered keys.

translation table. In the 4700 controller, a customer-selected relationship between a set of numeric codes and a set of alphameric characters and special characters. An input translation table translates input from a keyboard into alphameric data for the controller. An output translation table translates alphameric data into codes understood by the display monitor or printer.

verification code. A sixteen-bit value that serves as an "alias" for the controller's 56-bit master key.

Bibliography

The publications listed below contain information that may be useful to persons installing a 4700 system that includes cryptographic facilities.

IBM Vocabulary for Data Processing Telecommunication and Office Systems, GC20-1699

IBM System/370 Bibliography, GC20-0001

IBM System/370 Bibliography of Industry Systems and Application Programs, GC20-0370

IBM 4700 Finance Communication System:

System Summary, GC31-2016

Subsystem Operating Procedures, GC31-2032

Subsystem Problem Determination Guide, GC31-2033

Host Support User's Guide, SC31-0020

4701 Controller Operating Instructions, GC31-2022

4704 Display Operating Instructions, GC31-2025

Online Terminal Support for System/34, Program Description and Operation, SC31-0023

IBM 3624 Consumer Transaction Facility, Programmer's Guide, GC66-0088

IBM 3624 Consumer Transaction Facility, Programmer's Reference and Component Descriptions, GC66-0009

IBM System/34 Interactive Communication Feature, Reference Manual, SC21-7751

IBM Cryptographic Subsystem, Concepts and Facilities, GC22-9063

IBM 3848 Cryptographic Unit Product Description and Operating Procedures, GA22-7033

OS/VS Cryptographic Unit Support General Information Manual, GC28-1015

OS/VS Cryptographic Unit Support Installation Reference Manual, SC28-1016

OS/VS Programmer Cryptographic Facility General Information Manual, GC28-0942

OS/VS Programmed Cryptographic Facility Installation Reference Manual, SC28-0956

Systems Network Architecture (SNA) General Information, GA27-3102

Index

Key	Symbol	Used to Protect	Enciphered Under	For Use By
===	======	===============	==========	=======
ncrypting Keys				
ter Key	KM	KS, KMAC		
st Variant	KM1	KCD1, KCDs, KP2		
ond Variant	KM2	KCD2, KCDr		
rd Variant	KM3	KP1, KPv		
ıss–Domain Keys (in Cryptographic Storage)				
Sending	KCD1	KS	eKM1(KCD1)	RFMK, 330–1–3
Receiving	KCD2	KS	eKM2(KCD2)	RTMK, 330–1–4
ıss–Domain Keys (in Program Storage)				
Sending	KCDs	KS	eKM1(KCDs)	RFMK
Receiving	KCDr	KS	eKM2(KCDr)	RTMK
·Encrypting Keys				
sion Key	KS	Data	eKM(KS)	ENCIPHER, DECIPHER
sage Auth– tication Key	KMAC	Data	eKM(KMAC)	MACGEN
ncrypting Keys				
ut PIN Pro– ction Key	KP1	PINs	eKM3(KP1)	PINTRANS, PINVERIF
put PIN Pro– ction Key	KP2	PINs	eKM1(KP2)	PINTRANS
Validation y	KPv	PINs	eKM3(KPv)	PINVERIF

-1. Summary of 4700 Cryptographic Keys

0	Principal	Principal	Parameter
uction	Inputs	Outputs	List(s)
======	=========	=========	=========
HER	eKM(K) Data	eK(Data)	ENCPAR

HER	eKM(K)
	eK(Data)
		Data	ENCPAR

N ntext		K	KYGPAR
N phered		eKM(K)	KYGPAR
g KCD1)	eKM(K)	eKCD1(K)	RMKPAR

g KCDs)	eKM(K)
	eKM1(KCDs)
		eKCDs(K)	RMKPAR

g KCD2)	eKCD2(K)	eKM(K)	RMKPAR

g KCDr)	eKCDr(K)
	eKM2(KCDr)
		eKM(K)	RMKPAR

RIF	eKP1(PIN)
	eKM3(KP1)
	eKM3(KPv)
	Data
		Valid or Not-Valid indication	INPPAR
			VER3PAR

ANS	eKP1(PIN)
	eKM3(KP1)
	eKM1(KP2)
		Reenciphered and/ or reformatted PIN	INPPAR
			TNPPAR

N	eKM(KMAC) Data	MAC Data	ENCPAR
E	K Data	eK(Data)	
E	K eK(Data)	Data	

-2. Summary of 4700 Cryptographic Instructions

4700 Finance Communication System
Controller Programming Library
Volume 5
Cryptographic Programming
Order No. GC31-2070-0

READER'S COMMENT FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate. Comments may be written in your own language; English is not required.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*
Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address, and date:

____

____

____

____

What is your occupation?______________________________

Number of latest Newsletter associated with this publication:____________________

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

GC31-2070-0

Reader's Comment Form

Fold and Tape | Please Do Not Staple | Fold and Tape

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 78C
1001 W.T. Harris Boulevard
Charlotte, NC, USA 28257

Fold and Tape | Please Do Not Staple | Fold and Tape

4700 Finance Communication System Controller Programming Library Volume 5
Cryptographic Programming (File No. S370/4300/8100/S34-30) Printed in U.S.A. GC31-2070-0

File Number
S370/4300/8100/S34-30

Publication Number
GC31-2070-0

Printed in
USA