EMV Transactions with the ID TECH Universal SDK

Rev. C

August 6, 2018

ID TECH

10721 Walker Street, Cypress, CA90630 Voice: (714) 761-6368 Fax: (714) 761-8880

Visit us at http://www.ID TECHproducts.com

The information contained herein is provided to the user as a convenience. While every effort has been made to ensure accuracy, ID TECH is not responsible for damages that might occur because of errors or omissions, including any loss of profit or other commercial damage, nor for any infringements or patents or other rights of third parties that may result from its use. The specifications described herein were current at the time of publication, but are subject to change at any time without prior notice.

LIMITED WARRANTY

ID TECH warrants to the original purchaser for a period of 12 months from the date of invoice that this product is in good working order and free from defects in material and workmanship under normal use and service. ID TECH's obligation under this warranty is limited to, at its option, replacing, repairing, or giving credit for any product that returned to the factory of origin with the warranty period and with transportation charges and insurance prepaid, and which is, after examination, disclosed to ID TECH's satisfaction to be defective. The expense of removal and reinstallation of any item or items of equipment is not included in this warranty. No person, firm, or corporation is authorized to assume for ID TECH any other liabilities in connection with the sales of any product. In no event shall ID TECH be liable for any special, incidental or consequential damages to purchaser or any third party caused by any defective item of equipment, whether that defect is warranted against or not. Purchaser's sole and exclusive remedy for defective equipment, which does not conform to the requirements of sales, is to have such equipment replaced or repaired by ID TECH. For limited warranty service during the warranty period, please contact ID TECH to obtain a Return Material Authorization (RMA) number & instructions for returning the product.

THIS WARRANTY IS IN LIEU OF ALL OTHER WARRANTIES OF MERCHANTABILITY OR FITNESS FOR PARTICULAR PURPOSE. THERE ARE NO OTHER WARRANTIES OR GUARANTEES, EXPRESS OR IMPLIED, OTHER THAN THOSE HEREIN STATED. THIS PRODUCT IS SOLD AS IS. IN NO EVENT SHALL ID TECH BE LIABLE FOR CLAIMS BASED UPON BREACH OF EXPRESS OR IMPLIED WARRANTY OF NEGLIGENCE OF ANY OTHER DAMAGES WHETHER DIRECT, IMMEDIATE, FORESEEABLE, CONSEQUENTIAL OR SPECIAL OR FOR ANY EXPENSE INCURRED BY REASON OF THE USE OR MISUSE, SALE OR FABRICATIONS OF PRODUCTS WHICH DO NOT CONFORM TO THE TERMS AND CONDITIONS OF THE CONTRACT.

CONTENTS

Introduction

Chip-card (EMV) transaction processing is significantly more complex than magstripe transaction processing. More steps are involved, and there is considerable two-way back-and-forth communication between terminal and reader. Understanding all of the security, data packaging, messaging, and other considerations that apply to EMV transactions can be daunting.

ID TECH's Level 2 EMV kernel takes care of many of the low-level details involved in carrying out EMV transactions. The developer's job is made even easier by ID TECH's Universal SDK, which provides high-level-language access to library routines that can greatly speed EMV development. The Universal SDK (available for iOS, Android, and Windows) gives developers a rich API and a robust set of tools (and source-code examples) for communicating with ID TECH's EMV-capable card readers, greatly facilitating the creation, testing, and debugging of EMV applications using languages like C#, Java, Swift, or Objective-C.

> NOTE: To obtain the Universal SDK, please visit the Downloads page at https://atlassian.idtechproducts.com/confluence/display/KB/Downloads+-+Home. On that page, you will find links to special builds of the SDK for each supported ID TECH product and platform. (You'll also find user manuals and a variety of other helpful supporting documents.) No registration is necessary. All downloads are free and unlocked.

ID TECH's Universal SDK comes with source code for rich, fully featured demo apps, showing how the ID TECH EMV Level 2 kernel can be leveraged to create secure, high-performance applications with minimal effort. Nevertheless, knowing "where to begin" can be a challenge. So with that in mind, this document will look at some of the more important aspects of handling EMV transactions, and how the Universal SDK can help.

Section 1 of this document (How Does an "EMV Transaction" Work?) gives a quick semi-technical overview of what happens in an "EMV transaction." Most of the low-level interactions discussed in that section happen automatically, in ID TECH devices, at the kernel level; we present the information merely for background. If you have not already familiarized yourself with how EMV transactions work, take time to understand this section.

Section 2 of this document (EMV Transactions Using the Universal SDK) talks about EMV process flow in the context of a custom application that communicates with an ID TECH device. In that section, we'll have frequent opportunity to talk about Universal SDK libraries (and associated methods) and what *you* have to do to get an EMV transaction to work.

1 How Does an "EMV Transaction" Work?

This section assumes that the device in question is carrying out a *contact* EMV transaction. If you're familiar with EMV 4.3 specifications, this will be a quick review of things you probably already know. If you haven't studied the EMV specifications, you should do so. What follows is not a substitute for reading the actual specs.

The best way to think of an EMV transaction is as a conversation between an ICC (chip) card and a terminal, with the card reader acting as an intermediary. *Most* of the back-and-forth communication that occurs during an EMV transaction actually occurs between the card and the reader's Level 2 (or L2) kernel. The terminal (i.e.,

the host device to which the card reader is connected) is notified of the outcome of various stages of the conversation. Depending on the types of notifications received, the terminal might need to display certain messages on a display device, prompt the cardholder for more information, or (in some cases) do nothing. Most of the kernel-level operations that happen require no action on the terminal's part. For more information on what the terminal's responsibilities are at various points in the transaction, see the section on EMV Transaction Flow.

In the meantime, let's look at what goes on during an EMV transaction.

At a high level, the EMV transaction can be broken down into the following phases:

- ICC Power Up (ATR)
- Application Selection
- Data Authentication
- Processing Restrictions
- Version Check
- Application Usage Control
- Date Checks
- Cardholder Verification
- Terminal Risk Management
- Floor Limit Checking
- Velocity Checking
- Terminal Action Analysis
- Card Action Analysis
- Issuer Script Processing
- Completion

To keep track of these steps, the reader maintains two checklists. One is called the “Terminal Verification Results” or TVR. And the other one is called the “Transaction Status Information” or TSI. Both checklists start as arrays of bits set to 0 (two bytes' worth for TSI, five bytes for TVR).

During the conversation between the card and the terminal, the reader will check off certain items on its TVR checklist in to make sure it doesn’t do something unwarranted, like accept a fraudulent payment. At the same time, the reader will check off items on its TSI checklist to keep track of where the transaction is in the process.

At the end of a transaction, the TSI and TVR data are returned in TLV tags 9B and 95, respectively.

Let's take a quick look at each of the major steps of an EMV transaction, before turning out attention to ID TECH's Universal SDK.

1.1 ICC Power Up (ATR)

The card reader will begin a transaction by asking the card if it's awake, essentially, resulting in an Answer to Reset (ATR). The Answer To Reset (ATR) is a message output by a Smart Card conforming to ISO/IEC 7816 standards, given in response to electrical reset of the card's chip by a card reader. The presence of an ATR is often used as an indication that a Smart Card is operative, and the content of the ATR response can be examined as a first test that the card is of the appropriate kind for a given usage.

> NOTE: Failure of a chip card to respond properly to ATR can trigger fallback to MSR.

If the card responds appropriately to the ATR, a communication session will be set up between the card and the reader, using either the T0 protocol or the T1 protocol, as defined in ISO/IEC 7816-3.

1.2 Application Selection

An EMV card may be configured for multiple payment applications. After the card is powered up (and communication is established), an application needs to be selected. (Here, "application" means a card-resident application, not a terminal-resident payment app.) The process for doing this is somewhat involved, since the card reader maintains its own list of pre-loaded application identifiers (or 'AIDs'), which need to be matched against the application(s) available on the card. The selection process involves the reader querying the card to discover which applications are available on the card. Eventually, through a well-determined procedure specified by EMVCo, the card and reader "decide" on an application to use.

> **Note**: At runtime, application selection occurs at the kernel level. You normally will not have to write your own code for this.

It's possible the card's applications might not match any of the AIDs for which the terminal was certified, in which case the process will stop with an error of "no matching AID." This can trigger fallback to MSR. The EMV process can continue only if the reader and card agree on the application (and AID) that will be used.

Once a suitable application is selected, the card will respond with a Processing-options Data Object List (PDOL). The PDOL is a list of data elements that the card needs to receive from the terminal in order to execute the next command in the transaction process. In essence, the terminal (or card reader, in this case) needs to know what EMV functionality the card supports, and where the information is (on the chip) to support that functionality. To obtain the necessary information, the reader queries the card via APDUs conforming to ISO-7816. This is a very low-level type of interaction that you will ordinarily not have to concern yourself with (since the kernel handles it automatically); however, if you should need to interact with a card at this level, the ID TECH Universal SDK does expose APDU-related methods that enable this kind of interaction.

1.3 Data Authentication

In the previous step, the reader will have obtained an Application Interchange Profile (AIP) indicating the types of functions supported by the card. The AIP tells which type of Offline Data Authentication (ODA) the card supports.

There are three types of ODA:

- Combined Dynamic Data Authentication (CDA)
- Dynamic Data Authentication (DDA)
- Static Data Authentication (SDA)

Industry (EMV) specifications spell out the procedure for determining which kind of ODA must be used under a given set of conditions. The details aren't important here, since (again) this is a low-level operation that happens at the kernel level.

Depending on the outcome of ODA processing, various bits will be set in the TVR and TSI.

1.4 Processing Restrictions

Sometimes a card will be restricted for use in a specific country, or for a specific service; or a card may be expired.

During the “processing restrictions” step the terminal checks to see:

- Whether the application version on the card is the same as on the terminal
- Whether the type of transaction is allowed
- Whether the card is valid and not expired

Again, these are kernel checks; they do not require that you write any code yourself.

1.5 Version Check

The card and the reader (if it's certified by a card issuer) both have an application version number for a given application. This number is prescribed by the payment scheme (for example, MasterCard). As set forth by EMV specifications, the reader checks the applicable application version numbers to see if they agree. Depending on the outcome of this step, a bit in the TVR will be set.

1.6 Application Usage Control

In the application selection process, the reader will have received an Application Usage Control (AUC) record. This 2-byte bit array will tell the reader whether the card:

Is valid for domestic cash transaction
Is valid for international cash transaction
Is valid for domestic goods
Is valid for international goods
Is valid for domestic services
Is valid for international services
Is valid at ATMs
Is valid at terminals other than ATMs
Allows domestic cashback
Allows international cashback

The reader checks whether the transaction it is processing is allowed by the AUC or not. (This is a kernel check. You do not have to write code for it.) If the transaction is *not* allowed, the ‘Requested service not allowed for card product’ bit in the TVR is set to 1.

1.7 Date Checks

Often, a card will be issued that is not yet valid at the moment of issuing. This information can be set on the card in the Application Effective Date record.

If the card has an Application Effective Date and it is after the current date, the ‘Application not yet effective’ bit in the TVR is set. Otherwise nothing is set.

Applications have an expiration date. The card exposes the Application Expiration Date to the reader. If this date is in the *future*, the transaction continues normally. Otherwise the ‘Expired Application’ bit in the TVR is set to 1.

1.8 Cardholder Verification

The card issuer specifies various ways in which a cardholder may be allowed to prove that he or she is the rightful holder of the card. These methods of proof are called Cardholder Verification Methods (or CVMs).

The CVMs allowed by EMV can (at the issuer's option) include:

- Online PIN
- Offline Enciphered PIN
- Offline Plaintext PIN
- Signature
- No-CVM

The process for determining how a particular CVM is selected and how it works is somewhat involved. (See the EMV specifications for details.) The most important thing to note is that *some* type of CVM is generally performed (except for certain low-transaction-amount scenarios, in which "No-CVM" might apply), and the results of the CVM processing will set appropriate bits in the TVR and TSI. For example, the TVR contains bits that can record:

- Cardholder verification was not successful
- Unrecognized CVM
- PIN Try Limit exceeded
- PIN entry required and PIN pad not present or not working
- PIN entry required, PIN pad present, but PIN was not entered
- Online PIN entered

Once the “cardholder verification” step has been run, the “Cardholder verification was performed” bit in the TSI will be set to 1.

1.9 Terminal Risk Management

Obviously, the objective of terminal risk management is to protect the payment system from fraud. The risk of fraud is generally smaller when the terminal requests online approval from the issuer for a transaction. But online approval is not always *required*.

To determine whether the transaction should go online, the terminal will check three things:

- If the transaction is above the offline floor limit (see next section)
- Whether it wants to randomly select this transaction to go online
- Whether the card has had, or not had, an online authorization in a while

Once this step has been performed, the ‘Terminal risk management was performed’ bit in the TSI is set to 1.

1.9.1 Floor Limit Checking

If the value of the transaction is above the floor limit set in the terminal, the “Transaction exceeds floor limit” bit in the TVR is set to 1.

Note that a terminal may randomly select a transaction for online processing. If the transaction is selected, the ‘Transaction selected randomly for online processing’ bit in the TVR is set to 1.

1.9.2 Velocity Checking

If a card has not been online in a while, it may be that the current usage attempt is fraudulent. To mitigate the attendant risk, a card may have a Lower Consecutive Offline Limit (LCOL) and a Upper Consecutive Offline Limit (UCOL) set.

If the LCOL and UCOL have been provided to the payment device, it must do *velocity checking*.

The device will first request the Application Transaction Counter (ATC) and the Last Online ATC Register. The ATC is a counter that is incremented by one every time a transaction is performed. The Last Online ATC Register is set to the value of the ATC when a transaction has been online. The difference between the two is the number of transactions that have been performed offline.

- If the difference is higher than the LCOL, the 'Lower consecutive limit exceeded' bit in the TVR will be set to 1
- If the difference is also higher than the UCOL, the 'Upper consecutive limit exceeded' bit in the TVR is also set to 1
- If the Last Online ATC Register is 0, the "New card" bit in the TVR will be set to 1

1.10 Terminal Action Analysis

The payment device must decide whether to decline the transaction, complete it offline, or complete it online. At this point, it is only a preliminary decision. The device has to *ask the card* for confirmation of its decision. The card's chip will issue its advice based on the data available in a data object list. The chip's decision (which might override the reader's initial advice) will be issued during the "card action analysis" step (below).

Both the payment device and the card have settings that determine the action to take based on the TVR.

The settings on the card are called Issuer Action Codes (IAC). The settings on the device are called Terminal Action Codes (TAC).

There are three IACs and three TACs:

- TAC/IAC Denial
- TAC/IAC Online
- TAC/IAC Default

Just as with the TVR, these action codes are 5-byte bit arrays. These arrays are logically OR'd together at runtime (and compared with the TVR) to determine a course of action. The details aren't important here, since this is handled automatically, in the kernel, in accordance with EMV requirements.

Regardless of the decision taken by the device, it has to request confirmation from the card. (The card may disagree with the terminal.) The payment device requests confirmation by using the Generate Application Cryptogram (or "Generate AC") command. With this command, it will request to either: *decline, approve offline*, or *approve online*.

Along with this request, the terminal will provide the card with the required data for the Card Action Analysis step (see below). When the terminal asks the card to perform the Card Action Analysis, this is called the Generate Application Cryptogram request, or "first Gen AC."

Comment: *In most EMV transactions, there are two Gen AC requests. This means the card is typically asked for advice twice: once before going online, and once after the online decision has been obtained. In some parts of the world where offline transactions are allowed, the second Gen AC request might not need to be issued if the first Gen AC produced a TC cryptogram (meaning the transaction was "approved offline"). Likewise, if the first Gen AC resulted in an AAC cryptogram ("declined"), there is no need to continue to a second Gen AC. See below.*

1.11 Card Action Analysis

In this step, the card will perform its own risk management checks. *How* the card does this is outside the scope of EMV. The card only needs to communicate the results of its decision. This result is communicated using a *cryptogram*, which in this case is a digitally signed hash of transaction-related data. The issuer will typically use the cryptogram and the data in it to confirm that the card is authentic and that the proper risk management has been performed.

The card will generate one of three possible cryptograms:

- Transaction approved: Transaction Certificate (TC)
- Request online approval: Authorization Request Cryptogram (ARQC)
- Transaction declined: Application Authentication Cryptogram (AAC)

At the conclusion of this step, the card provides a TC, ARQC, or AAC cryptogram to the payment device (in Tag 9F26) together with the transaction data that were used to generate the cryptogram. The terminal will set the “Card risk management was performed” bit in the TSI to 1.

Comment: *Since it is impossible to tell, by looking at the cryptogram itself, what kind of cryptogram it is, information about the type of cryptogram can be found in the top four bits of data in the 9F27 tag. If the top "nibble" is 00, the cryptogram was of type AAC ("declined"). If the top nibble is 40 (hex), the cryptogram was of type TC ("approved"). A value of 80 means it was of type ARQC ("go online").*

If the card generates a TC, the transaction is automatically approved offline, but the terminal needs to provide the cryptogram to the Issuer in order to capture the funds of the transaction. If the card gives an ARQC, the terminal will need to request authorization online. (This is the expected outcome in many scenarios, particularly in the U.S., which is essentially an online-only or at least predominantly online payment environment.) If the terminal received a TC or AAC, the transaction is now finished (with an offline approval or offline decline, respectively).

1.11.1 Online Processing & Issuer Authentication

After the first Gen AC request, the card may provide an ARQC, indicating that the terminal or payment app should go online for authorization. This step is the responsibility of the payment app, rather than the card reader. It usually means contacting a gateway, acquirer, or other network endpoint where the request can be relayed to an Issuer or other authority. The decision of the Issuer is then relayed back to the gateway and originating app.

The processing done by the Issuer during this step is technically outside the scope of EMV, but at the very least, the issuer will attempt to authenticate the card by validating the ARQC cryptogram. In addition, the issuer will generally perform its own risk management and check if the cardholder has sufficient credit or funds.

The Issuer will respond with either an approval or decline code. The issuer is expected to generate a response cryptogram using data known to the card. The card can use this data to verify that the response received is really from the issuer.

The card will already have told the payment device whether or not it supports issuer authentication (this information is given in the AIP). If a response cryptogram was received and the card *supports* issuer authentication, the terminal will request authentication using the EXTERNAL AUTHENTICATE command. If the resulting issuer authentication fails, the “Issuer authentication failed” bit in the TVR will be set to 1.

Once issuer authentication has been performed, the “Issuer authentication was performed” bit in the TSI is set to 1.

1.11.2 What Happens If the Terminal Cannot Go Online?

In case of power outage, network outage, etc., it may not be possible to go online. In this case, the transaction continues, but the Completion step will produce an 'AAC' cryptogram ("Declined"). This does not mean the transaction cannot be submitted for later settlement. A "Decline" by the chip card is merely the *advice* of the card; the final decision on whether a transaction can be honored always rests with the issuer.

> **NOTE:** In "online" markets (like the U.S.), gateways usually specify Stand-In Processing (STIP) procedures for merchants to follow in cases where network outage (or other problems) preclude going online. Every acquirer has its own rules for performing STIP. You should obtain, understand, and adhere to the STIP rules laid out in your acquirer's implementation guide (or integrator's guide). Consult that guide for details.

1.12 Issuer Script Processing

In some instances, the Issuer may need to update data on the card. This can be done using issuer script processing.

> **Comment:** Depending on whether scripts were specified in Tag 71 or 72, script processing may occur before (71) or after (72) the second Gen AC request.

If issuer scripts were processed, the card reader will set the “Script processing was performed” bit in the TSI to 1. Also:

- If issuer script processing *fails* before the second Generate AC command, the “Script processing failed before final GENERATE AC” bit in TVR will be set to 1.
- If the issuer script processing fails *after* the second generate AC command, the “Script processing failed after final GENERATE AC” bit in TVR will be set to 1.

1.13 Completion

If the terminal went online for approval and a response was received, the terminal will request a final transaction cryptogram using the Generate AC command for a second time. (This is the so-called "second Gen AC.") As before, the cryptogram itself will come in tag 9F26, and information about the *type* of cryptogram will come in tag 9F27.

At completion time, the card can only respond with a TC or AAC. The TC is required in order to capture the funds from the issuer. AAC is typically associated with a Decline. (Note that in EMV, the card can disagree with the online advice and issue an AAC even if the online authority approved the transaction.) The card will typically issue an AAC if the terminal provided a 'Z3' response code in Tag 8A (indicating "unable to go online").

Note that the card's advice is not necessarily binding, for purposes of completing a transaction. That is to say, a final cryptogram result of AAC doesn't mean the transaction cannot subsequently be posted for settlement. In Faster EMV or Quick Chip, for example, an AAC is *always* returned at the second Gen AC (since the card is

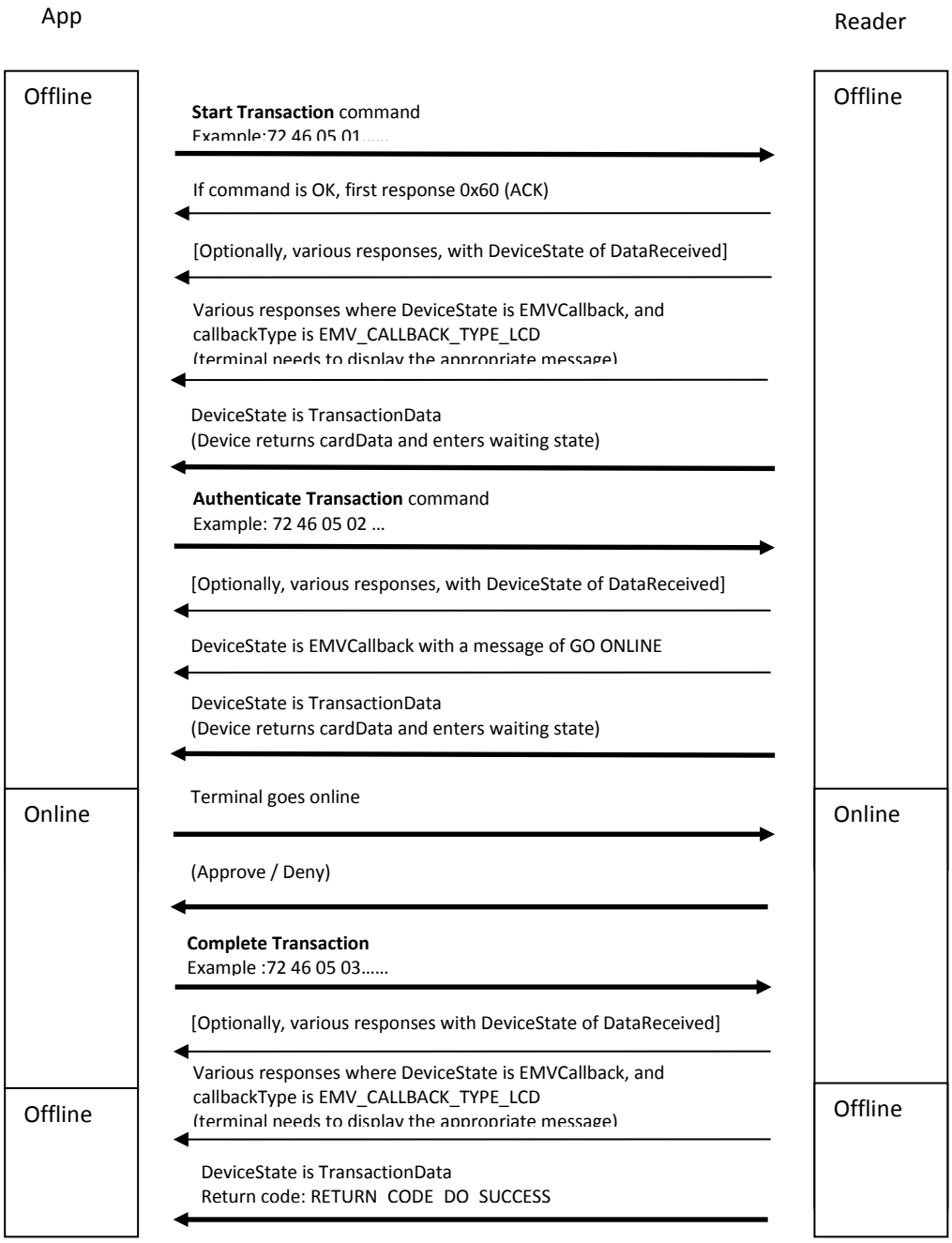
routinely told the terminal could not go online; and the card is bound to issue an AAC after the 'Z3'). An AAC at this point in the transaction is merely advisory: It's the card's *advice*. The issuer can overrule it, and often will.

2 EMV Transactions using the Universal SDK

For the integrator writing applications for ID TECH devices using the Universal SDK, the good news is that almost all of the operations described above are *kernel-level* operations that happen automatically, requiring no custom code. Extra code might come into play if you need to customize various aspects of the steps outlined above or configure the device in a certain way. To help with this, the ID TECH Universal SDK exposes a variety of methods for obtaining low-level access to features of the kernel that are normally "out of sight" (such as AIDs and CAPKs), so that you can override or update various features as needed.

> **Note:** ID TECH's Universal SDK comes with source code for two apps. One of the apps represents the project created using the Tutorial in the documentation. (It contains "Simple_Demo" in the file name.) The other demo app is more elaborate (and contains "SDKDemo" in the name). In the sections below, references to the "demo app" apply to the latter project, not the Tutorial project.

At a high level, conducting an EMV transaction on an ID TECH device results in a flow that looks like this (see diagram below):



The above diagram shows a typical transaction. From the developer's point of view, the transaction is really a three-part process, involving calls to asynchronous SDK methods `emv_startTransaction()`, `emv_authenticateTransaction()`, and `emv_completeTransaction()`. During each phase, the reader carries out various operations at the kernel level, then calls the application back via the callback registered using `IDT_Device.setCallback(MessageCallBack)`. Messages received from the reader can be inspected to determine what the terminal needs to do during each phase (such as process exceptions, display messages on a screen, or go online).

In order to perform EMV transactions using the Universal SDK, you will first need to initialize your card reader (e.g., Augusta) via the methods outlined below; then you can run transactions using the appropriate transaction methods, namely `emv_startTransaction(), emv_authenticateTransaction(),` and `emv_completeTransaction().` In the next section, we'll take a look at the initializations you'll need to do.

2.1 Configuration

Before attempting to perform any EMV transactions using the Universal SDK, you will want to be sure your card reader is properly configured. At a minimum, this means setting:

- Terminal Configuration values
- Default AIDs (Application Identifiers)
- CAPKs (Certificate Authority Public Keys)

If you bypass these steps, you will encounter errors when trying to do an EMV transaction.

The Universal Demo program (for Windows) contains built-in commands, and a graphical UI, to assist you in performing these setup steps. However, the SDK also allows you to do configuration tasks programmatically. In the sections that follow, we talk about programmatic configuration using the C# version of the Universal SDK.

After you configure the foregoing items, the settings will be persistent across device reboots and you will not have to do them again. CAPKs, however, may sometimes go out of date and require refreshing. (AIDs rarely, if ever, change.) Contact your processor's or Card Brand's relationship manager to obtain the latest CAPKs and learn about their expiration cycles.

2.1.1 Terminal Configuration

Prior to performing EMV transactions, you will need to set certain terminal configuration parameters that will apply to all transactions, using `SharedController.emv_setTerminalData()`.

To accomplish this one-time configuration operation, you might have a private method called `mySetTerminalData()` that looks something like this:

```
private void mySetTerminalData(object sender, EventArgs e)
{
        byte[] term = Common.getByteArray("
        5f3601029f1a0208409f3501229f330360f8c89f4005f00000a0019f1e085465726d6
        96e616c9f150212349f160f303030303030303030303030303030309f1c0838373635343332319f4e22313
        03732312057616c6b65722053742e20437970726573732c204341202c5553412edf260101df10086e56e6
        67265737a68df110101df270100dfee150101dfee160100dfee170107dfee180180dfee1e08f0dc3cf0c
        29e9400dfee1f0180dfee1b0830303031353130dfee20013cdfee21010adfee2203323c3c");

        RETURN_CODE rt = IDT_MiniSmartII.SharedController.emv_setTerminalData(term);

        if (rt == RETURN_CODE.RETURN_CODE_DO_SUCCESS) {
                tbOutput.AppendText("Set Terminal Successful:" + " nrnn");
        }
        else {
                log("Save Terminal failed Error Code: " + "0x" + String.Format("{0:X}",
                (ushort)rt) + ": " + IDTechSDK.errorCode.getErrorString(rt) + "nrnn");
        }
}
```

Here, the byte array `term` contains TLV information for "Transaction Currency Exponent" (tag 5F36), "Terminal Country Code" (tag 9F1A), "Terminal Type" (tag 9F35), "Terminal Capabilities" (tag 9F33), "Additional Terminal Capabilities (tag 9F40), "Interface Device Serial Number" (9F1E), "Tag Application Label" (tag 50), "Directory Discretionary Template" (tag 73), and "Issuer Script Command" (tag 86).

> **Comment:** *The terminal configuration step (see code above) is typically a one-time setup operation. It does not need to be done on a per-transaction basis.*

ID TECH readers can support at least five major kernel configurations (1C through 5C). The choice of configuration is guided by, among other things, the type of Cardholder Verification Method supported by the scenario:

- Configuration 1C is for attended devices using chip and PIN (e.g. ID TECH VP8800).
- Augusta supports 2C (which allows cardholder confirmation of Language Selection and application selection) and 5C (no customer confirmations). For Quick Chip, use 5C.
- The default configuration for Spectrum Pro is 4C, which supports "no CVM" (only).
- 3C adds chip and PIN capabilities with the use of a PIN pad (e.g., ID TECH SmartPIN L100). Neither 3C nor 4C allow for chip-and-signature. (Chip-and-signature wouldn't make sense for Spectrum Pro, which is an unattended device.)

Many terminal capability settings (which are configured by means of TLVs, as indicated above) are considered "minor" settings that can be tailored at will to a given scenario. Some are considered "major" settings that can't be changed without producing a runtime error. For more information on Terminal Settings and kernel configurations (and the settings that can be changed without producing an error), be sure to consult the Knowledge Base article at the following URL:
https://atlassian.idtechproducts.com/confluence/pages/viewpage.action?pageId=32276534.

2.1.2 Setting AIDs

The card reader needs to know which application identifiers (AIDs) you intend to support. This is something you will configure using code similar to the following:

```
       RETURN_CODE rt;
       byte[] name = Common.getByteArray("a0000000031010");
       byte[] aid =
Common.getByteArray("9f01065649534130305f5701005f2a0208409f090200965f3601029f1b0400003a98df
25039f3704df28039f0802dfee150101df13050000000000df14050000000000df15050000000000df180100df1
70400002710df190100");

       // Tell Augusta the name and value of the AID you wish to support:
       rt = IDT_Augusta.SharedController.emv_setApplicationData(name, aid);

       if (rt == RETURN_CODE.RETURN_CODE_DO_SUCCESS)
            {
                tbOutput.AppendText("Default AID Successful\r\n");
            }
       else
            {
                tbOutput.AppendText("Default AID failed Error Code: " +
                      "0x" + String.Format("{0:X}", (ushort)rt) +
                      ":" + IDTechSDK.errorCode.getErrorString(rt) +
                      "\r\n");
            }
```

Consult the Universal SDK's demo app source code for more detail. (The demo app source code shows how to load AIDs for all the commonly supported card applications.)

As with Terminal Configuration, setting the default AIDs is something you will likely do only once, not on a per-transaction basis.

2.1.3 Setting CAPKs

You will also want to set the reader's supported Certificate Authority Public Key values. (Each card brand has one or more CAPKs for doing Offline Data Authentication.) The code for doing this looks like:

```
        byte[] capk =
Common.getByteArray("a000000003500101b769775668cacb5d22a647d1d993141edab7237b000100018000d1
1197590057b84196c2f4d11a8f3c05408f422a35d702f90106ea5b019bb28ae607aa9cdebcd0d81a38d48c7ebb0
062d287369ec0c42124246ac30d80cd602ab7238d51084ded4698162c59d25eac1e66255b4db2352526ef0982c3
b8ad3d1cce85b01db5788e75e09f44be7361366def9d1e1317b05e5d0ff5290f88a0db47");

        RETURN_CODE rt = IDT_Augusta.SharedController.emv_setCAPK(capk);
```

This (again) is something you will likely configure once, not on a per-transaction basis.

> TIP: If your EMV transactions are failing because of inability to perform ODA (as indicated in the first byte of TVR data, tag 95), check to be sure your CAPKs are up to date.

With initializations out of the way, it's time to do an EMV transaction. Let's look at the phases of the EMV transaction, one by one.

2.2 Start Transaction

This command, issued via the method `emv_startTransaction()`, does the following:

When a card is already seated:

- Initiates a Power On sequence with the ICC reader
- If an ICC cannot be detected (no ATR:Answer To Reset), then a fallback condition occurs and a swipe is requested.
- If ATR received, establishes timeout for receipt of “Authenticate Transaction” command and transmits “Application Data” parameters to the card, and the EMV transaction starts.

When a card is not seated:

- The MSR Swipe is enabled, along with monitoring for a card to be inserted.

When a Swipe is executed first:

- If a non-ICC card is swiped, the swipe data is returned and the transaction is over.
- If a ICC card is swiped, the swipe is rejected and the kernel prompts to use the ICC (insert the card).

Note that if you wish to inspect ATR response parameters, you may optionally issue an ICC power-on via `SharedController.icc_powerOnICC(ref ATR)` *before* calling `emv_startTransaction()`, although this is strictly optional.

When executing `emv_startTransaction()`, TLV data should, at a minimum, contain the following tags:

- 9F02: Amount
- 9F03: Other amount
- 9C: Transaction type

In addition, you can (optionally) include a DFEE1A tag to request what tags be included in the final response callback. The DFEE1A tag is a proprietary ID TECH tag that wrappers other tags. For example, to request that tags 9F36, 9F37, and 95 be included in the output from the Authenticate Transaction stage, specify DFEE1A059F369F3795 as part of the TLV byte array. (The 05 after DFEE1A is the total length of the included tags, 9F36, 9F37, and 95.) Please be aware that this will override the default set of default tags returned. (See the discussion under Obtaining Extra Tags, further below.)

Another tag that can be used is DFEF1F, which causes the Authenticate Transaction phase to be entered automatically (without a call to `emv_authenticateTransaction()`, after Start Transaction has finished. To use this tag, supply two bytes of data: a first byte (with 1 to signal auto-authenticate, or 0 to signal no auto-authenticate) and a second byte that, if set to a value of 1, will force the transaction to go online. (Zero is the default.) The complete TLV might look something like DFEF1F020100, where 02 is the length.

After the Start Transaction command is issued, the reader will interact with the card to determine the correct AID to use, and carry out other low-level EMV operations. It will call the application back a minimum of two times (once with ACK, and once with TransactionData), but there will typically be additional status callbacks as well. You can inspect the `DeviceState` object (the second argument to the callback) to determine whether the reader is responding with device data, an EMV callback, or transaction data. (Code for this is shown in the sample demo app that comes with the Universal SDK.)

With a typical ID TECH EMV L2 device (e.g., Augusta), the first response from the reader, in raw form, might look like 02010006060603, which is just an ACK (0x06) wrapped in a header and trailer. The header in this case is STX (0x02) plus two length bytes, whereas the trailer consists of an LRC value, checksum, and ETX (0x03). Since the message is just ACK (0x06), the LRC is 0x06 and the checksum is 0x06.

When the reader calls back with an `EMVCallback`, you should check the callback type, and if it is `EMV_CALLBACK_TYPE_LCD`, have the terminal display the appropriate message on the device LCD or POS screen. (Code showing how to do this is given in the sample demo app.) If action is required from the customer before proceeding, the payment app should prompt the customer as necessary (using the terminal UI) and collect any needed info before proceeding. See the demo app's `processEMVCallback()` method for an example of how to handle `EMVCallback` messages.

When the reader responds with a `DeviceState` of `TransactionData` and an EMV result of `EMV_RESULT_CODE_AUTHENTICATE_TRANSACTION`, the transaction can go to the next step. At this stage, the reader will respond only to three commands: “Authenticate EMV L2 Transaction” , “Cancel EMV L2 Transaction”, or “Retrieve EMV L2 Transaction Data,” or (in other words) the SDK methods `emv_authenticateTransaction()`, `emv_cancelTransaction()`, and `emv_retrieveTransactionResult()`. The reader will block, at this point, while your app decides which method to call. (Eventually, it will time out, if no action is taken.) You can use this opportunity to inspect the available transaction data to determine whether to invoke any extra logic that might be appropriate to the transaction. (For example, this might be when the app decides that the customer is eligible for a loyalty discount.) If no special logic applies, simply proceed to the next step: Authenticate Transaction.

2.3 Authenticate Transaction

In this stage of the transaction, the card reader's EMV kernel will attempt to apply cardholder verification methods, terminal risk management logic, and any other logic required under EMV processing rules, in order to determine whether the terminal needs to go online for authorization.

The purpose of this stage is to allow the temporary interruption of program flow so that you can (optionally) modify/change any existing tags. For example, a common use of this stage is to determine (via business logic) whether the cardholder is entitled to a discount of some kind, and if so, provide the discount by changing the previously submitted amount (Tag 9F02) to a lesser amount. Any tags that need their values changed can be passed as a TLV stream in this method's parameter list.

Often, this step of pausing the EMV transaction to evaluate collected card data is not needed or utilized. You can tell the reader to continue automatically (with no pausing) by passing tag DFEF1F with a value of 0100 (or 0101 if force-online is requested), when executing the `emv_startTransaction()` method. This can be done per-transaction. But also note, the SDK has an "autoAuthenticate" option that is ON by default. If autoAuthenticate is set to ON before the EMV transaction begins, the authenticate transaction step will *always* occur without pausing. The main class has a method to set this Boolean: `emv_autoAuthenticate(bool authenticate)`.

The app should (as usual) monitor the `DeviceState` via the second argument of the `MessageCallBack` method (the method that was registered with the device at the time the app was loaded). Direct inspection of the `DeviceState` will tell you whether the callback is being called with device data, or with a state of `EMVCallback` (requiring a change of UI message, or perhaps interaction with the user), or with transaction data.

The final callback will have a `DeviceState` (second argument) of `TransactionData` and will be accompanied by a standard EMV result code conforming to one of the following:

```
public enum EMV_RESULT_CODE
    {
        EMV_RESULT_CODE_APPROVED_OFFLINE = 0,
        EMV_RESULT_CODE_DECLINED_OFFLINE = 1,
        EMV_RESULT_CODE_APPROVED = 2,
        EMV_RESULT_CODE_DECLINED = 3,
        EMV_RESULT_CODE_GO_ONLINE = 4,
        EMV_RESULT_CODE_CALL_YOUR_BANK = 5,
        EMV_RESULT_CODE_NOT_ACCEPTED = 6,
        EMV_RESULT_CODE_FALLBACK_TO_MSR = 7,
        EMV_RESULT_CODE_TIMEOUT = 8,
        EMV_RESULT_CODE_GO_ONLINE_CTLS = 9,
        EMV_RESULT_CODE_AUTHENTICATE_TRANSACTION = 16,
        EMV_RESULT_CODE_SWIPE_NON_ICC = 17,
        EMV_RESULT_CODE_CTLS_TWO_CARDS = 122,
        EMV_RESULT_CODE_CTLS_TERMINATE_TRY_ANOTHER = 125,
        EMV_RESULT_CODE_CTLS_TERMINATE = 126,
        EMV_RESULT_CODE_UNABLE_TO_REACH_HOST = 255,
        EMV_RESULT_CODE_FILE_ARG_INVALID = 4097,
        EMV_RESULT_CODE_FILE_OPEN_FAILED = 4098,
        EMV_RESULT_CODE_FILE_OPERATION_FAILED = 4099,
        EMV_RESULT_CODE_MEMORY_NOT_ENOUGH = 8193,
        EMV_RESULT_CODE_SMARTCARD_OK = 12289,
        EMV_RESULT_CODE_SMARTCARD_FAIL = 12290,
        EMV_RESULT_CODE_SMARTCARD_INIT_FAILED = 12291,
        EMV_RESULT_CODE_FALLBACK_SITUATION = 12292,
        EMV_RESULT_CODE_SMARTCARD_ABSENT = 12293,
        EMV_RESULT_CODE_SMARTCARD_TIMEOUT = 12294,
        EMV_RESULT_CODE_MSR_CARD_ERROR = 12295,
        EMV_RESULT_CODE_PARSING_TAGS_FAILED = 20481,
        EMV_RESULT_CODE_CARD_DATA_ELEMENT_DUPLICATE = 20482,
```

```
    EMV_RESULT_CODE_DATA_FORMAT_INCORRECT = 20483,
    EMV_RESULT_CODE_APP_NO_TERM = 20484,
    EMV_RESULT_CODE_APP_NO_MATCHING = 20485,
    EMV_RESULT_CODE_MANDATORY_OBJECT_MISSING = 20486,
    EMV_RESULT_CODE_APP_SELECTION_RETRY = 20487,
    EMV_RESULT_CODE_AMOUNT_ERROR_GET = 20488,
    EMV_RESULT_CODE_CARD_REJECTED = 20489,
    EMV_RESULT_CODE_AIP_NOT_RECEIVED = 20496,
    EMV_RESULT_CODE_AFL_NOT_RECEIVED = 20497,
    EMV_RESULT_CODE_AFL_LEN_OUT_OF_RANGE = 20498,
    EMV_RESULT_CODE_SFI_OUT_OF_RANGE = 20499,
    EMV_RESULT_CODE_AFL_INCORRECT = 20500,
    EMV_RESULT_CODE_EXP_DATE_INCORRECT = 20501,
    EMV_RESULT_CODE_EFF_DATE_INCORRECT = 20502,
    EMV_RESULT_CODE_ISS_COD_TBL_OUT_OF_RANGE = 20503,
    EMV_RESULT_CODE_CRYPTOGRAM_TYPE_INCORRECT = 20504,
    EMV_RESULT_CODE_PSE_BY_CARD_NOT_SUPPORTED = 20505,
    EMV_RESULT_CODE_USER_LANGUAGE_SELECTED = 20512,
    EMV_RESULT_CODE_SERVICE_NOT_ALLOWED = 20513,
    EMV_RESULT_CODE_NO_TAG_FOUND = 20514,
    EMV_RESULT_CODE_CARD_BLOCKED = 20515,
    EMV_RESULT_CODE_LEN_INCORRECT = 20516,
    EMV_RESULT_CODE_CARD_COM_ERROR = 20517,
    EMV_RESULT_CODE_TSC_NOT_INCREASED = 20518,
    EMV_RESULT_CODE_HASH_INCORRECT = 20519,
    EMV_RESULT_CODE_ARC_NOT_PRESENCED = 20520,
    EMV_RESULT_CODE_ARC_INVALID = 20521,
    EMV_RESULT_CODE_COMM_NO_ONLINE = 20528,
    EMV_RESULT_CODE_TRAN_TYPE_INCORRECT = 20529,
    EMV_RESULT_CODE_APP_NO_SUPPORT = 20530,
    EMV_RESULT_CODE_APP_NOT_SELECT = 20531,
    EMV_RESULT_CODE_LANG_NOT_SELECT = 20532,
    EMV_RESULT_CODE_TERM_DATA_NOT_PRESENCED = 20533,
    EMV_RESULT_CODE_CVM_TYPE_UNKNOWN = 24577,
    EMV_RESULT_CODE_CVM_AIP_NOT_SUPPORTED = 24578,
    EMV_RESULT_CODE_CVM_TAG_8E_MISSING = 24579,
    EMV_RESULT_CODE_CVM_TAG_8E_FORMAT_ERROR = 24580,
    EMV_RESULT_CODE_CVM_CODE_IS_NOT_SUPPORTED = 24581,
    EMV_RESULT_CODE_CVM_COND_CODE_IS_NOT_SUPPORTED = 24582,
    EMV_RESULT_CODE_CVM_NO_MORE = 24583,
    EMV_RESULT_CODE_PIN_BYPASSED_BEFORE = 24584
}
```

(Note that this list is subject to change. Consult the most recent Universal SDK documentation for the latest list.)

If the result code at the end of the Authenticate Transaction stage is `EMV_RESULT_CODE_GO_ONLINE`, then the terminal app should go online to obtain final authorization. (Going online is the responsibility of the payment app, not the reader. The EMV kernel has no knowledge of online endpoints, protocols, or APIs.)

At the conclusion of Authenticate Transaction, the application should consider the transaction finished if the cryptogram received from the card was of type 'TC' ("Approved") or 'AAC' ("Declined"). Otherwise, if the cryptogram was 'ARQC', the application should go online for authorization, then call `emv_completeTransaction()`.

> NOTE: To determine what kind of cryptogram was received in tag 9F26, inspect the top nibble of the CID in 9F27. A hex value of 00 indicates 'AAC'; 40 indicates 'TC'; 80 indicates 'ARQC'.

2.4 Complete Transaction

After going online for authorization, you must call `emv_completeTransaction()`. You must call this method even if you were unable to go online but still wish to proceed with the transaction. (You would do this, for example, if the network was down or the gateway failed to respond.)

You must provide the `emv_completeTransaction()` method with the parameters that tell it if you were able to successfully reach the approver. (These parameters should be provided in tag 8A. Typical values here are 0x3030 for approval, 0x3032 for referral, 0x3035 for decline, and 0x5A33 for "unable to go online.") Issuer Authentication Data (tag 91) and Issuer Scripts (tags 71 or 72) are not always present, but if they were returned by the online approver, they should be passed to this method. Consult the SDK documentation for the method signature and calling conventions; also consult the sample demo app code for an actual example of how to use this method.

When the transaction is complete, the reader will invoke your `MessageCallBack` method with a `DeviceState` of `TransactionData`. The sample demo app code shows how to pull TLV (tag) data out of the `IDTTransactionData` object provided in the fourth argument to the callback. Typically, this tag data will include TVR and TSI data, in tags 95 and 9B, respectively.

Tags from the reader may contain masked and encrypted data. ID TECH encrypts some (but not all) tag data, using criteria described in ID TECH document 80000502-001, *ID TECH Encrypted Data Output*. The SDK has methods that will parse a TLV stream as necessary when the card data is returned, so manual parsing of the raw TLV data should not be required. Indeed, this is one advantage of using the Universal SDK: No matter whether your data is TLV data or magstripe data, the SDK has methods to parse it for you.

2.5 TLV Data

EMV transactions produce TLV data (data conforming to the BER-TLV tag/length/value convention; see Annex B, Book 3, EMV 4.3). At any stage in a transaction, you can use the `emv_unencryptedTags` field of your `IDTTransactionData` object (which has an instance name of `cardData` in the SDK sample code) to obtain unencrypted tags, or the `emv_encryptedTags` field to obtain encrypted tags. These fields point to byte[] arrays that you then need to process further, if you wish to obtain actual TLVs. For example, you can use a routine like the following to convert the byte arrays to a Dictionary from which you can obtain text versions of the TLVs:

```
private string tlvToValues(byte[] tlv) {

            string text = "";
            Dictionary<string, string> dict =
                   Common.processTLVUnencrypted(tlv);
            foreach (KeyValuePair<string, string> kvp in dict)
                   text += kvp.Key + ": " + kvp.Value + "\r\n";
            return text;
}
```

2.5.1 Default Tags

Each stage of the transaction produces TLV data specific to the stage in question. The default TLVs obtained at each stage will vary somewhat by product, but the following are typically returned:

Start Transaction
4F
50
57
5A
5F20
5F24
5F25
5F2D
5F34
84
9F20
DFEE12
DFEE23

Authenticate Transaction
95
9B
9F02
9F03
9F10
9F13
9F26
9F27
9F34
9F36
9F37
9F4D
9F4F

Complete Transaction
95
99
9B
9F02
9F03
9F10
9F13
9F26
9F27
9F34
9F36
9F37
9F4D
9F4F
9F5B

Note that almost all of these are EMVCo-defined standard tags. ID TECH proprietary tags (typically three bytes in length, starting with DFEE or DFEF) sometimes also appear. For example, DFEE12 contains the transaction's KSN (Key Serial Number).

For a complete listing of ID TECH proprietary tags and their meanings, refer to document 80000503-001, *ID TECH TLV Tag Reference Guide,* available for download at https://atlassian.idtechproducts.com/confluence/display/KB/Downloads+-+Home.

2.5.2 Encrypted Tags

Some TLVs will contain encrypted data, if your device has been key-injected (and has encryption turned on). Generally speaking, any tags containing track data (56, 57, 9F6B) or PAN data (5A), or any other data considered sensitive, will be encrypted according to the rules spelled out in document 80000502-001-F, *ID TECH Encrypted Data Output*. (This document can be downloaded from the above web page.)

By default, the following tags (if present in your data) will be encrypted, assuming you are using a device that has been key-injected, with encryption enabled:

5A
56
57
9F1F
9F20
9F6B
FFEE13
FFEE14
DF812A
DF812B
DF31
DF32

2.5.3 Obtaining Extra Tags

Recall that you can (optionally) notify the card reader to produce extra tags by including a DFEE1A tag in your original call to `emv_startTransaction()`. See Start Transaction, further above. To obtain extra tags, first include a DFEE1A TLV in your call to `emv_startTransaction()` and/or `emv_authenticateTransaction()`, and/or `emv_completeTransaction()`. Then, after each phase has finished, you must call `emv_retrieveTransactionResult()` to retrieve the requested TLVs.

> **NOTE:** You must call `retrieveTransactionResult()` within 15 seconds after the transaction is over. For security reasons, this data is not persisted in memory for an extended period of time after the transaction completes.

IMPORTANT: When you specify additional tags in `emv_startTransaction()`, the tags you specify are the *only* tags you can expect to retrieve. The default set of EMV tags will be overridden by your request for other tags.

Also note: Tags 57 and 5A are available after `emv_startTransaction()` by default (as indicated further above), but are *not* included by default after `emv_authenticateTransaction()`, or

`emv_completeTransaction()`. As a rule, you should collect the TLVs you want immediately after each phase of the transaction.

3 Contactless EMV Transactions

So far, we've been talking about contact EMV transactions involving a chip card inserted into a "contact" slot. But some ID TECH readers (particularly those sold under the ViVOpay brand) also include contactless EMV capability, allowing transactions to be initiated via cell phone and/or a "proximity card" (credit card with RFID capability). What follows is a very brief overview of how to conduct contactless transactions using the Universal SDK.

3.1 Configuration

As with contact EMV, contactless EMV requires that you first configure the reader with terminal settings, CAPKs, and AIDs before undertaking any transactions. These configuration items are *not* shared with "contact EMV" configurations; rather, you must set them up separately for contactless. Use the Universal SDK's `ctls_setApplicationData()` and `ctls_setCAPK()` methods to load each AID and CAPK that your payment app needs to support. (You will do this configuration once, and then the settings will persist across device reboots.)

"Terminal settings" are a bit more complicated in contactless EMV than in contact EMV, because in contactless EMV there are, in effect, multiple EMV kernels: one kernel per card brand. (Even this is a bit of a simplification.) The upshot is that you will need to provide different terminal settings for different card brands. In contactless readers, "terminal settings" are specified by Groups of TLVs. Predefined Groups (with "Group numbers," such as 00, 80, 90, A0, B0, and C0) exist, containing configurable TLVs that allow a set of behaviors to be mapped to a particular card or set of cards. For example: Group 00 ("group zero") contains around three dozen TLVs that apply specifically to transactions that involve Visa Debit/Credit, Visa Interlink, Visa Electron, and/or JCB cards. The settings in Group 00 might not be appropriate for other cards, however, so (for example) MasterCard has its own group: Group 80. Likewise, American Express uses the configurations supplied in Group A0.

> **NOTE:** *Groups are numbered somewhat differently in ID TECH's older "NEO I" devices than in later "NEO II" devices. Also, NEO I devices (such as VP3300 and Kiosk III) used many two-byte TLVs (e.g. FFE4) that have since been superseded by three-byte tags (e.g. DFEE2D). If you are not sure which type of device you have, or which NEO Interface Developer's Guide (IDG) applies to your device, ask your ID TECH representative for guidance.*

ID TECH has predefined a number of Groups that are populated/set by default whenever the firmware command '04-09' is issued. In the Universal SDK, you can run the 04-09 command using the `device_sendVivoCommandP2()` method, with '04' as the command argument and '09' as the subcommand argument. Once you run this command, all Groups are initialized (for all card brands) with common defaults.

You may need to override certain defaults, of course. For example, you might need to set your Currency Code (tag 5F2A) to something other than '0840' if you're not in the U.S. To override terminal settings, you need to provide the TLVs you want to override, along with a specified Group number, to the `ctls_setConfigurationGroup(byte[] tlv)` method. The first TLV in the byte array should be the Group number (specified using the FFE4 tag in NEO I, or DFEE2D in NEO II); then at least one TLV should follow, containing configuration information. For example, in a NEO I device, you could set Group 4 to have a Currency Code of 978 (for Euros) by supplying the following TLV data (as a byte array):

```
FF E4 01 04 9F 03 06 00 00 00 00 00 00 9A 03 FF FF FF 9F 21 03 FF FF
FF 9C 01 00 5F 2A 02 09 78 9F 1A 02 08 40 9F 33 03 00 08 E8 9F 09 02
00 01 FF FD 05 00 00 00 00 00 FF FE 05 00 00 00 00 00 FF FF 05 00 00
00 00 00 9F 6E 04 D8 E0 00 00 DF 51 01 C0 FF EE 1C 04 00 00 00 3C 9F
```

```
1B 04 00 00 17 70 FF F1 06 00 00 00 01 50 00 FF F506 00 00 00 00 50
00 9F 35 01 22
```

For clarity, tags are shown in blue, lengths in orange, and data in brown. Note that the first tag is FFE4, which has 04 as the data value (meaning, all of the downstream TLVs should be applied to Group 4). Even though we are interested only in changing tag 5F2A to 0978 (highlighted in yellow), we supply all the TLVs for this group, so that no values are dropped and no values get picked up from Group 0 by accident. (Any TLVs that don't exist in your Group, but that *do* exist in the default Group 00, will get picked up from Group 00 unless you override the default value explicitly.)

> **IMPORTANT:** When editing Group 00 (the default group, which cannot be deleted), you can replace individual tags without affecting other tags, but for all other Groups, the `ctls_setConfigurationGroup(byte[] tlv)` method overwrites *all* data and installs only the TLV(s) that were passed in (as a byte array). Therefore, the best practice to follow, even if you are only modifying one TLV, is to read *all* TLVs from the Group in question, using `ctls_getConfigurationGroup(int group, ref byte[] tlv);` then modify the tag(s) of interest; then replace *all* TLVs. But also remember that any tags you fail to supply to a custom group, if they already exist in Group 00, will inherit values from Group 00 by default.

A full discussion of Groups and their management is beyond the scope of this document. You can find more information in the relevant NEO IDG (or the AR 3.0.0 IDG, if you're developing for VP8800). You can also consult the appropriate Appendix in the IDG to learn what the default TLV values are for each group. Or, use the `ctls_getAllConfigurationGroups(ref byte response[][])` method to obtain all TLVs for all Groups, at any time.

3.2 Starting a Transaction

At the firmware-command level, you can start a contactless transaction, in any NEO device, with '0240' (command 02 and subcommand 40), or, equivalently, you can issue the SDK method `ctls_startTransaction (double amount, double amtOther, int exponent, int type, int timeout, byte[] tags, bool forceOnline).` You can use the same callback routine that you defined for contact EMV (see EMV Transactions using the Universal SDK further above). When your callback is called, you can inspect the `IDTechSDK.IDTTransactionData` instance (in the final argument of the callback) to determine what kind of transaction occurred, and many other types of info. (Consult the SDK documentation for an exhaustive list of properties.)

Contactless EMV transactions can be initiated upon command ("Poll on Demand" Mode), or the reader can be set to auto-detect NFC/RFID cards ("Auto Poll" Mode), and you can set the reader to either mode using `device_setPollMode( byte mode)`, where mode can be 0 for Auto Poll or 1 for Poll on Demand. When your callback is triggered after Auto Poll detects a card, you can call `device_getTransactionResults (ref IDTTransactionData results)` to obtain the results of the transaction.

When issuing `ctls_startTransaction()` in Poll on Demand Mode, you can tell the device to accept contactless presentations only, *or* you can tell it to accept all three types of presentations (MSR, contact, *and* contactless). To specify the type(s) of presentations you want to accept, use tag DFEF37 (as shown below); specify fallback support using DFEF3C.

DFEF37	01	Define the type of interface to be activated with 02-40 Interface Select: Bit 0: MSR Bit 1: Contactless Bit 2: Contact	DF EF 37 01 07 07 = 0000 0111 This activates transaction for all 3 interfaces.

DFEF3C	03	Fallback support and Timeout value for waiting for the next command (mainly to support EMV workflow) Byte 1: Fallback support Byte 2~3: Timeout for next command (Unit: Sec)	DF EF 3C 03 01 00 60 Fallback is supported, and the timeout is set to 60 seconds before the transaction times out.

Example raw command (NEO firmware instructions):

```
5669564f74656368320002400022 1e9c01009f02060000000001009f0306000000
000000dfef370107dfef3c03010060 18d1
```

To send this command via the SDK, use the `device_sendVivoCommandP2()` method with a command of '02', a subcommand of '40', and tag data (as byte array) of 9c01009f02060000000001009f0306000000000000dfef370107dfef3c03010060.

NOTE: Unlike contact EMV, a *contactless* transaction runs straight through to completion, without interruption, in one step. You will get *one* cryptogram, in tag 9F26, at the end, rather than two cryptograms (corresponding to the two Gen AC events of contact EMV). A contactless transaction occurs atomically, in one swift step (lasting about 500 milliseconds). There is no `ctls_completeTransaction()` method, because completion is automatic. If the transaction occurs without error, all of the TLVs you need (including clearing-record TLVs) will be in the returned data.

4 Additional Resources

You can find a variety of technical resources (white papers, SDKs, demos, utilities, documentation) on EMV-based application development at the ID TECH Knowledge Base: https://atlassian.idtechproducts.com/confluence/display/KB/EMV+tools+and+reference+-+downloads. No registration is required. Downloads are free and unlocked.

On the Knowledge Base, you'll find ID TECH document 80000502-001, *ID TECH Encrypted Data Output*. This document is invaluable for understanding how ID TECH readers package MSR and EMV data.

A contact-EMV sample application with source code can be found at https://atlassian.idtechproducts.com/confluence/download/attachments/30479625/Contact%20EMV%20Demo%201.00.zip?api=v2. This Windows-based app, written in C#, illustrates the programming idioms commonly used in EMV payment applications that take advantage of the Universal SDK.

You can find builds of the Universal SDK for C# on Windows, Java on Android, and Swift on iOS, as well as a C++ version for Linux. The latter comes with a Java JNI binding, in case you need to support the use of native code using Java on Linux.

Be sure to consult our Knowledge Base for documentation and other resources: https://atlassian.idtechproducts.com/confluence/display/KB/Knowledge+Base+-+Home.

And don't hesitate to reach out to us with questions. Drop us a note at support@idtechproducts.com to open a support ticket automatically. You'll hear back from us within one business day.

Revision History

Rev.	Date	Change	Author
50	9/19/2016	Initial draft.	KT
A	11/16/2016	Add Initializations section (and sub-sections on Terminal Configuration, AIDs, and CAPKs), with code examples. Include discussion of obtaining extra tags.	KT
	1/3/2017	Add internal links.	KT
	2/17/2017	Add section on Stand-In Processing (unable to go online). Expand discussion of Gen AC choreography.	KT
B	8/10/2017	Add explicit listing of encrypted tags. Add "Additional Resources" section. Expand discussion of Terminal Configurations. Add reference to contact-EMV sample app. Various clarifications.	KT
C	8/2/2018	Miscellaneous corrections. Add Contactless EMV section. Reformat.	KT
	8/3/2018	Bug fixes, clarifications.	