EPAS Client Library

API specification v1.2

Contents

1 Document Notes

1.1 History

1.1.1 Version 1.0, 05/05/2021

Initial release

1.1.2 Version 1.1, 20/05/2021

Added details on handling refunds of alternative payment methods, see section 9.8.

1.1.3 Version 1.2, 06/07/2021

- Added details on serial / USB connection, see section 7.2.
- Added details on OperatingParameters.StorageDirectory, see section 4.1.1.
- Clarified use of GenerateSimpleReceipt, see section 12.1.17.
- Moved from .Net Standard 2.0 to .Net Framework 4.7.2.

1.2 Glossary

Term	Meaning
DCC	Dynamic Currency Conversion
ECR	Electronic Cash Register
EPAS	Electronic Protocol Application Software

1.3 References

1. EPASOrg: “Sale to POI Protocol Specifications”, v1.0, October 2010
2. Swedbank / CEKAB: “Cardholder and Operator Interface”, v4.02, August 2009
3. Swedbank bulletin: “Contactless”, v1.0 release 5

2 Introduction

2.1 What is EPAS

The EPAS protocol (document ref 1) is defines a large and complex protocol for the interaction of payment terminals and ECRs.

Westpay payment terminals implement a subset of this specification, but also some customisations and specialisations. This is because the world of payments and terminals has moved on substantially since 2010 and the documented protocol was not capable enough to do what was required.

2.2 The EPAS Client Library

The combination of the complexity of EPAS and the Westpay customisations meant that integrating an ECR with a Westpay terminal was not a simple process, and there was an obvious need to have a client library that would make it much simpler to integrate an ECR with a Westpay payment terminal.

The EPAS Client Library is intended to greatly simply the task of ECR developers who are integrating their product with Westpay terminals.

The features of the EPAS Client Library are:

- Implemented as a .Net 4.7.2 module for easy integration.
- Provides a simple interface to operate the full range of supported transactions.
- Removes the need to know anything of the EPAS protocol.
- Removes the need to implement connectivity to the terminal.

3 Integrating the EPAS Client Library

The steps to integrate the EPAS Client Library are:

1. Add the library module to the ECR project.
2. Add an instance of the library to the ECR application.
3. Implement required methods.
4. Implement optional handler methods as required.

Once this framework is in place the library can be used to run transactions.

3.1 Creating a library instance

The EPAS Client Library is implemented by the EpasClient class in the Westpay.Epas namespace.

The client takes a single parameter, which is an instance of a class that implements IClientApp. The methods that must be implemented for IClientApp are detailed in section 5.

3.2 Calling API methods

API calls into the library are always synchronous. When a purchase transaction is called, for example, the method does not return until the transaction has ended. In general, therefore, it is best for the ECR software to avoid calling API methods from the user interface thread.

3.3 Handler methods

The EPAS Client Library works with a variety of callback methods. Some of these are required by the interface definition, while some are optional. Section 5 deals with the required interface methods, while the optional callbacks are detailed in “Optional handler methods” on page 21.

When the library calls methods in the ECR, only one method will be called at any time. This is done to ensure that there are no race conditions, since events can happen quickly and it would be confusing if, for example, the *CardAccepted* handler is called while the *CardInserted* handler is still executing.

It is, therefore, recommended that the ECR returns from interface methods and handlers as quickly as possible so that the remainder of the transaction can be processed without delay.

There is one exception to this, which is the *Log* method of the IClientApp interface. This has a dedicated thread so that the *Log* method can be called while another method or handler is executing. Only one call to *Log* will be made at a time, however – there will be no overlapping calls made.

3.4 Return values

Most library methods return an *ApiResult* value. These are related to the operation of the library, the communications link, and the protocols involved. They are not directly related to transactions or other high-level operations.

The return values are defined as:

ApiResult value	Meaning
OK	The request / message was successfully sent to the terminal for processing

ErrorInvalidParameter	An invalid parameter was supplied to the method called
ErrorConnectionFailure	The library failed to connect to the terminal
ErrorNotConnected	The library is not yet connected to the terminal
ErrorCommunicationsFailure	The library was unable to send the request / message to the terminal
ErrorLibraryFailure	An internal error occurred in the library
ErrorParseFailure	The response from the terminal could not be parsed

Note: a return value of ApiResult.OK only indicates that the low level communications and messaging was successful. It does not, for example, indicate that a transaction was approved.

3.5 Receipt printing

The EPAS implementation in the terminal provides the receipt contents are separate data elements, rather than as a pre-formatted block of text. The reasons for this include:

- Receipt printers are not limited to fixed-width character sets, so aligning text elements is not a simpler matter of adding spaces.
- Even if using a fixed-width font, receipt printer widths vary from device to device.
- A pre-formatted receipt does not allow for receipt elements to be printed in different fonts, e.g. larger or bold text to show the transaction type and amount.
- Merchants may wish to use electronic receipts, and therefore need access to individual data elements rather than a pre-formatted receipt.

The terminal will issue separate receipts for the customer and merchant, where required. Each receipt is provided by a call to the *PrintReceipt* method, covered in section 5.2.

3.6 Log storage

The EPAS Client Library will receive logging data from the terminal. This is stored in a series of files named “Tracelog.txt”, “Tracelog_1.txt”, up to “Tracelog_9.txt”. These files are automatically rotated and limited in size. “Tracelog.txt” is the most recent file.

These log files will be useful if there is any unexpected behaviour in the terminal, e.g. a failed transaction or a problematic card.

These files are stored by default in the user’s temporary folder. On Windows 10 this is C:\Users*<username>*\AppData\Local\Temp.

The client can specify a more convenient location by modifying the value of *StorageDirectory* in the library’s *Options* property, e.g.

```
string storage =
        Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments),
        "EpasClientLibraryLogs");
try
{
    Directory.CreateDirectory(storage);
    mEpas.Options.StorageDirectory = storage;
}
catch (Exception ex)
{
```

```
    Log("Unable to use " + storage + " for file storage: " + ex.Message);
}
```

This code fragment creates a directory under the users Documents folder and specifies that as the place for the library to store log files.

3.7 Dynamic Currency Conversion (DCC)

The terminal supports DCC. DCC permits the customer to pay an amount that is presented in their own currency, rather than the currency of the merchant.

DCC is only possible when:

- DCC has been configured in the terminal,
- A chip card is used,
- The currency on the chip card is different to the merchant's currency,
- The card range is enabled for DCC in the terminal,
- There is no cashback,
- The transaction is a purchase or refund,
- The card is not a 'national' card.

Refunds require special handling, as detailed below.

When DCC can apply in a purchase the terminal will make an online request to find out if DCC can be offered. If DCC is available then the cardholder is prompted to choose between the original amount in the merchant currency or the converted amount in their card's currency.

If they choose DCC then the transaction continues in the merchant's currency, i.e. the transaction currency does not actually change, but the transaction amount is adjusted so that the amount they see on their card statement is the amount that they were shown on screen.

DCC affects the ECR in two ways:

3.7.1 Disclaimer texts

DCC receipts require a DCC disclaimer to be printed on the receipt. The receipt data (see section 5.2) includes a DCC disclaimer text in English.

The disclaimer texts are, however, subject to change. If the DCC provider specifies a new disclaimer text, that change may take some time to be implemented in the terminal. The receipt data, therefore, also includes the individual data elements that allow an ECR to build its own DCC disclaimer text.

Disclaimer texts also change according to the card issuer, and according to the transaction type, and even according to who is authorising the transaction. The disclaimer text in the receipt data is valid at the point where the payment application is built, but the disclaimer texts can change over time according to rule changes so the ECR should be aware of its responsibilities here.

3.7.2 DCC and refunds

When a refund that is made to a card that is eligible for DCC, the ECR must indicate if DCC was performed on the original transaction. This is covered in section 5.9.

The intention here is that if a purchase was done using DCC then the refund should also be done using DCC, even though this may mean that the refunded amount may not be the same as the purchase amount, due to exchange rate changes in the time between the two transactions.

4 Initialisation

4.1 Initialising the EPAS client library

The EPAS client library has no initialisation method.

All that is required is to provide an implementation of IClientApp as the parameter to the initialiser, e.g.

```
using Westpay.Epas;

public partial class SampleECR :IClientApp
{
    /// <summary>
    /// Instance of the EPAS client library
    /// </summary>
    EpasClient mEpas = null;

    public SampleECR()
    {
        // Create the EPAS client library instance
        mEpas = new EpasClient(this);
    }

    .
    .
    .

}
```

The ECR application must then implement all the requirements of the IClientApp interface, as described in section 5.

4.1.1 Library configuration

There are some configuration options in the client library. These are defined in the library's **Options** property, which is an instance of the **OperatingParameters** class.

The properties of OperatingParameters are detailed below.

EnableTip

```
bool EnableTip
```

The ECR can set this to true to enable tipping or false to disable tipping. Tipping only applies to purchase transactions. Note that tipping functionality must also be enabled in the terminal configuration before it will be active. This setting is more useful to disable tipping when it is enabled in configuration.

Default value: false

TransactionCurrency

```
CurrencyCodes TransactionCurrency
```

This property defines the currency that will be specified in EPAS messaging. In normal use this must match the currency that is configured in the terminal.

It is possible, however, for the terminal to be configured to accept transactions in different currencies, in which case the transaction currency here must be one of those that are configured.

Default value: CurrencyCodes.SEK

WaitForCardRemoval

```
bool WaitForCardRemoval
```

The *WaitForCardRemoval* property controls whether the terminal will wait for an inserted chip card to be removed before it sends the transaction response back, or if it sends the response before the card is taken.

Setting this to false can make the transaction feel faster, since the result is available sooner. But setting this to true makes it less likely that the customer will leave their card in the terminal by mistake.

Regardless of this setting the terminal will not allow a new transaction until the card has been taken.

Default value: true

LogXmlMessages

```
bool LogXmlMessages
```

If set to true, the *LogXmlMessages* property causes the incoming and outgoing XML messages that make up the EPAS protocol to be logged with the ECR at debug level.

Default value: false

StorageDirectory

```
string StorageDirectory
```

The *StorageDirectory* property specifies a directory that the library can use to store files. Typically these will be logs from the terminal, but the usage may be extended in future updates.

See section 3.6 for more information on this.

5 Implementing IClientApp

IClientApp requires the implementation of several methods, as detailed below.

5.1 IClientApp.Display

```
void Display(string text);
```

5.1.1 Remarks

The terminal will send messages intended for the cashier. These are localised into the cashier's chosen language and are not always the same as the messages that appear on the terminal's display.

Each line in the text is separated by a newline character ("\n").

5.1.2 Example usage

```
/// <summary>
/// Handle cashier display requests from the library
/// </summary>
/// <param name="text">Text to display</param>
public void Display(string text)
{
    if (InvokeRequired)
    {
        Invoke(new Action(() => Display(text)));
    }
    else
    {
        txtCashierDisplay.Text = text.Replace("\n", Environment.NewLine);
    }
}
```

5.2 IClientApp.PrintReceipt

```
void PrintReceipt(ReceiptData receipt);
```

5.2.1 Remarks

Most transactions will require two receipts to be printed, one for the cardholder and one for the merchant. Each receipt is printed using a call to PrintReceipt and all the data required to produce the receipt is in the ReceiptData class instance that is passed as a parameter.

Some transaction will only involve a single receipt, e.g. transactions that are declined will only produce a cardholder receipt. It is also possible for receipts to be produced in other situations, e.g. if the terminal loses power during a transaction then it should attempt to print a receipt for the transaction that was in progress once connectivity has been restored.

Also, the receipts may not always be sent at the end of a transaction. If, for example, the transaction is being authorised by signature then one receipt, the one that is to be signed, will be produced earlier and the outcome of the transaction will then depend on whether that signature is correct or not.

Note: a receipt will only be printed after a card has been accepted. If a transaction is cancelled before a card is accepted then there will be no receipt.

Note: when PrintReceipt is called during a transaction then it is recommended to store the receipt data until the transaction is complete. The receipts can be printed when the method is called, the information in the receipt data will probably be useful for reporting once the transaction is complete.

Note: If a transaction is partly completed, e.g. at PIN entry, and the terminal loses power then when the terminal restarts PrintReceipt will be called after the ECR has logged in. The ECR must, therefore, be able to handle PrintReceipt outside of the context of an active transaction.

5.2.2 Data elements

The receipt data is held in a structure of different classes, enumerations, and properties. To make this document easier to follow, these are detailed separately in section 12.

5.2.3 Example usage

```
/// <summary>
/// Handle a receipt from the terminal
/// </summary>
/// <param name="rcpt">Receipt details</param>
public void PrintReceipt(ReceiptData rcpt)
{
    Log(LogLevel.Status, "Receipt");
    LogFixedWidth(rcpt.GenerateSimpleReceipt(32));
}
```

5.3 IClientApp.PrintReport

```
void PrintReport(string report);
```

5.3.1 Remarks

The terminal can produce reports, e.g. transaction summaries or configuration details. These are presented as a single string where each line is separated by a newline character (“\n”).

The ECR can, of course, choose to save these reports to disk or show them on screen rather than print them.

5.3.2 Example usage

```
/// <summary>
/// Handle a report from the terminal
/// </summary>
/// <param name="report">Report text</param>
public void PrintReport(string report)
{
    Log(LogLevel.Status, "Report");
    LogFixedWidth(report);
}
```

5.4 IClientApp.VerifySignature

```
bool VerifySignature(string displayText, out bool signatureOk);
```

5.4.1 Remarks

Cardholders may have the option to verify their identity using signature rather than by PIN entry. If signature verification is used then this method will be called. A text prompt for the cashier is provided in *displayText*.

If the signature is valid then the ECR should set *signatureOk* to true, otherwise *signatureOk* should be set to false.

Normally the ECR should return true from this method, indicating that the setting of *signatureOk* is valid. If the ECR returns false then the library will abort the current transaction.

The ECR can implement signature verification such that it always approves the signature (i.e. *signatureOk* is set to true and the method returns true) when this method is called, but then ensure that the signature is checked at the end of the transaction. The transaction can then be reversed if the signature is not valid.

5.4.2 Example usage

```
/// <summary>
/// Verify the cardholder signature
/// </summary>
/// <param name="displayText">Text to display</param>
/// <param name="signatureOk">True if the signature is valid</param>
/// <returns>True if handled, false to abort the transaction</returns>
public bool VerifySignature(string displayText, out bool signatureOk)
{
    Display(displayText);
    signatureOk = MessageBox.Show("Signature OK?", "Verify signature",
MessageBoxButtons.YesNo) == DialogResult.Yes;
    return true;
}
```

5.5 IClientApp.HandleVoiceReferral

```
bool HandleVoiceReferral(string displayText, out string approvalCode);
```

5.5.1 Remarks

If a transaction cannot be authorised online due to communications problems then it may be suitable for voice referral. This is where the cashier contacts the authorisation centre directly by telephone and provides the transaction and card details verbally. If the transaction is approved then the cashier will be provided with an approval code.

When this method is called the text prompt for the cashier is provided in *displayText*, and this should contain the phone number to call.

The approval code should be provided in the *approvalCode* parameter. If the value in *approvalCode* is empty or null on return then the transaction will be aborted.

Normally the ECR should return true from this method, indicating that the setting of *approvalCode* is valid. If the ECR returns false then the library will abort the current transaction.

5.5.2 Example usage

```
/// <summary>
/// Do a voice referral authorisation on the transaction
/// </summary>
/// <param name="displayText">Text to display</param>
/// <param name="approvalCode">Approval code from the authorising organisation</param>
/// <returns>True if handled, false to abort the transaction</returns>
bool IClientApp.HandleVoiceReferral(string displayText, out string approvalCode)
{
    bool result = false;

    approvalCode = null;

    TextEntry vr = new TextEntry("Voice referral approval code", displayText);
    if (vr.ShowDialog() == DialogResult.OK)
    {
```

```
        approvalCode = vr.EnteredText;
        result = true;
    }

    return result;
}
```

5.6 IClientApp.PaymentCodeRequired

```
bool PaymentCodeRequired(string displayText, out string paymentCode);
```

5.6.1 Remarks

Some cards can be configured such that they require a payment code to be entered. If this is the case then this method is called. The cardholder should know the payment code, and a text prompt for the cashier is provided in *displayText*.

The ECR should provide the payment code in the *paymentCode* parameter.

Normally the ECR should return true from this method, indicating that the setting of *paymentCode* is valid. If the ECR returns false then the library will abort the current transaction.

The payment application on the terminal will know what the expected payment code length will be. If the supplied code is not valid then the terminal may ask for the payment application to be re-entered.

5.6.2 Example usage

```
/// <summary>
/// Get a payment code from the cardholder
/// </summary>
/// <param name="displayText">Text to display</param>
/// <param name="paymentCode">Payment code provided</param>
/// <returns>True if handled, false to abort the transaction</returns>
public bool PaymentCodeRequired(string displayText, out string paymentCode)
{
    bool result = false;

    paymentCode = null;

    TextEntry pc = new TextEntry("Payment code", displayText);
    if (pc.ShowDialog() == DialogResult.OK)
    {
        paymentCode = pc.EnteredText;
        result = true;
    }

    return result;
}
```

5.7 IClientApp.VatAmountRequired

```
bool VatAmountRequired(string displayText, out decimal vatAmount);
```

5.7.1 Remarks

Some cards can be configured such that they require a VAT / sales tax amount to be provided. If this is the case then this method is called.

A text prompt for the cashier is provided in *displayText*. The VAT amount of the transaction should be provided in the *vatAmount* parameter.

Normally the ECR should return true from this method, indicating that the setting of *vatAmount* is valid. If the ECR returns false then the library will abort the current transaction.

5.7.2 Example usage

```
/// <summary>
/// Ask the cashier for the transaction's VAT amount
/// </summary>
/// <param name="displayText">Text to display</param>
/// <param name="vatAmount">Supplied VAT amount</param>
/// <returns>True if handled, false to abort the transaction</returns>
public bool VatAmountRequired(string displayText, out decimal vatAmount)
{
    bool result = false;

    vatAmount = 0;

    TextEntry vat = new TextEntry("VAT amount", displayText);
    if (vat.ShowDialog() == DialogResult.OK)
    {
        decimal.TryParse(vat.EnteredText, out vatAmount);
        result = true;
    }

    return result;
}
```

5.8 IClientApp.LoyaltyCardPresented

```
bool LoyaltyCardPresented(string cardNumber, out bool cardAccepted);
```

5.8.1 Remarks

The terminal has a list of card ranges and some of those identify cards as loyalty / bonus cards. If one of those cards is presented to the terminal then this method is called. The loyalty card number is provided in the *cardNumber* parameter.

The ECR should decide if the loyalty card number is valid for the merchant. If the card is valid then the ECR should set *cardAccepted* to true, otherwise false will indicate that the loyalty card was not accepted. Neither of these affects the transaction in progress on the terminal.

Normally the ECR should return true from this method, indicating that the setting of *cardAccepted* is valid. If the ECR returns false then the library will abort the current transaction.

5.8.2 Example usage

```
/// <summary>
/// Handle a loyalty card that has been presented in the terminal
/// </summary>
/// <param name="cardNumber">Loyalty card number</param>
/// <param name="cardAccepted">True if we can handle the card, false if not</param>
/// <returns>True if handled, false to abort the transaction</returns>
public bool LoyaltyCardPresented(string cardNumber, out bool cardAccepted)
{
    cardAccepted = MessageBox.Show("Accept loyalty card " + cardNumber, "Loyalty card",
MessageBoxButtons.YesNo) == DialogResult.Yes;
    return true;
}
```

5.9 IClientApp.CheckDccOnOriginalTransaction

```
bool CheckDccOnOriginalTransaction(string displayText, out bool dccWasUsed);
```

5.9.1 Remarks

Dynamic Currency Conversion is a process where a cardholder from a different country can choose to pay for a transaction in their home currency rather than in the currency of the merchant.

This introduces a challenge for refunds, because if a refund transaction is to be done with a 'foreign' card then the refund should be made using the converted currency if the original purchase was made with a converted currency, even though the currency rates may have changed between purchase and refund.

The only way to know if DCC was used in the original purchase is to check the receipt. If this method is called, therefore, the cashier should check the original receipt to find out if DCC was used. If so, *dccWasUsed* should be set to true, otherwise it should be false if DCC was not used.

The *displayText* parameter contains a prompt for the cashier to check for DCC on the original purchase.

Normally the ECR should return true from this method, indicating that the setting of *dccWasUsed* is valid. If the ECR returns false then the library will abort the current transaction.

5.9.2 Example usage

```
/// <summary>
/// Ask if DCC was used in the original purchase transaction
/// </summary>
/// <param name="displayText">Text to display</param>
/// <param name="dccWasUsed">True if DCC was used, false if not</param>
/// <returns>True if handled, false to abort the transaction</returns>
public bool CheckDccOnOriginalTransaction(string displayText, out bool dccWasUsed)
{
    Display(displayText);
    dccWasUsed = MessageBox.Show("Was DCC used in the original transaction?", "DCC",
MessageBoxButtons.YesNo) == DialogResult.Yes;
    return true;
}
```

5.10 IClientApp.ParameterDownloadAvailable

```
void ParameterDownloadAvailable(string displayText, out bool permitDownload);
```

5.10.1 Remarks

From time to time the terminal will need parameter updates to be done. These are usually done at the beginning of the day after the ECR logs in to the terminal, but it is possible that the terminal will ask the ECR if the parameters can be downloaded immediately. The ECR can choose to respond with *permitDownload* set to false to delay the parameter download until later, though the same question will be asked repeatedly.

It should be noted that this functionality is very rarely used. The simplest approach is to simply set *permitDownload* to true to allow the download, but the ECR could implement a time-based response that sets *permitDownload* according to the time of day.

The *displayText* parameter contains a prompt to inform the cashier that a download is available.

There is no return value here. In the other methods of this type, the return value indicates if the response is valid or if the transaction should be cancelled. This method, however, is only ever called after the end of a transaction so there is nothing to abort.

5.10.2 Example usage

```
/// <summary>
/// Ask if a parameter update should be done now, or delayed until later.
/// </summary>
/// <param name="displayText">Text to display</param>
/// <param name="permitDownload">True to permit the download, false to delay it</param>
public void ParameterDownloadAvailable(string displayText, out bool permitDownload)
{
    Display(displayText);
    permitDownload = MessageBox.Show("Download configuration?", "Configuration update",
MessageBoxButtons.YesNo) == DialogResult.Yes;
}
```

5.11 IClientApp.Log

```
void Log(LogLevel level, string text);
```

5.11.1 Remarks

The EPAS Client Library will generate logging in various situations. When this happens, this method is called.

The *level* parameter indicates that the log entry is one of error, status or debug logging. The text to be logged is in the *text* parameter.

If the ECR has a logging function then it is recommended that these texts are written to the ECR log. The ECR can, of course, filter the logging according to configuration. For example, it may not make sense to store debug level logging when in a production environment.

5.11.2 Example usage

```
/// <summary>
/// Provides a logging function that the library can use
/// </summary>
/// <param name="level">Logging level</param>
/// <param name="text">Text to display</param>
public void Log(LogLevel level, string text)
{
    try
    {
        LogTimestamp();
        RawLog(text);
    }
    catch (Exception)
    {
        // Logger exceptions are not logged.
    }
}
```

6 Optional handler methods

The client library offers a range of handler methods that are optional, and the ECR can implement any that will be useful.

Each handler method is implemented as a callback. The ECR simple assigns a method reference to the appropriate library callback handler and the method will be called when appropriate.

The handlers are documented below.

6.1 CardStatus

```
Void CardStatus(bool cardInserted)
```

6.1.1 Remarks

This callback is used when the terminal recognises that a card has been inserted in, or removed from, the card reader. The ECR might use this status to prompt the cardholder to remove the card when the transaction is complete.

This notification only applies to the smart card reader. Cards that are tapped or swiped do not trigger this callback.

Note that if a card inserted while the card readers in the terminal are not enabled then this notification is used as soon as the readers are enabled.

6.1.2 Example initialisation

```
mEpas.CardStatus = CardStatus;
```

6.1.3 Example usage

```
/// <summary>
/// Called when a card is inserted or removed
/// </summary>
///<param name="inserted">True if a card has been inserted, false if it has been
removed</param>
public void CardStatus(bool inserted)
{
    if (InvokeRequired)
    {
        Invoke(new Action(() => CardStatus(inserted)));
    }
    else
    {
        if (inserted)
        {
            lblCardPresent.Text = "CARD INSERTED";
        }
        else
        {
            lblCardPresent.Text = "CARD TAKEN";
        }
    }
}
```

6.2 CardAccepted

```
void CardAccepted(string maskedCardNumber)
```

6.2.1 Remarks

The terminal's configuration includes a list of card ranges that is used to identify valid card numbers. When a payment card is presented and is considered valid then this callback is used to notify the ECR of the masked card number. The masking means that only the first six digits and last four digits of the card number are shown. The rest are replaced with asterisk ('*') characters.

Note that loyalty cards are handled separately. This is covered in section 5.8, "IClientApp.LoyaltyCardPresented".

6.2.2 Example initialisation

```
mEpas.CardAccepted = HandleCardAccepted;
```

6.2.3 Example usage

```
/// <summary>
/// Called when a payment card is accepted by the terminal
/// </summary>
/// <param name="maskedCardNumber">Masked card number</param>
public void HandleCardAccepted(string maskedCardNumber)
{
    if (InvokeRequired)
    {
        Invoke(new Action(() => HandleCardAccepted(maskedCardNumber)));
    }
    else
    {
        lblCardPresent.Text = "CARD: " + maskedCardNumber);
        Log("Card accepted: " + maskedCardNumber);
    }
}
```

6.3 BusyStatus

```
void BusyStatus(bool busy)
```

6.3.1 Remarks

There are times when the terminal is going to be busy for an unknown time, normally when it is downloading new parameters. In these situations the terminal will notify its busy status by calling this callback with *busy* set to true. When the activity ends it will call this callback again with *busy* set to false.

6.3.2 Example initialisation

```
mEpas.BusyStatus = HandleBusyState;
```

6.3.3 Example usage

```
/// <summary>
/// Updates the ECR with the terminal's busy status
/// </summary>
/// <param name="busy">If true, the terminal is busy with something
/// that might take some time.  If false, the terminal is no longer
/// busy and can handle requests.</param>
public void HandleBusyState(bool busy)
{
    if (InvokeRequired)
    {
        Invoke(new Action(() => HandleBusyState(busy)));
    }
    else
```

```
    {
        if (busy)
        {
            lblBusy.BackColor = Color.LightPink;
            lblBusy.ForeColor = Color.DarkRed;
            lblBusy.Text = "BUSY";
        }
        else
        {
            lblBusy.BackColor = Color.Transparent;
            lblBusy.ForeColor = Color.Gray;
            lblBusy.Text = "OK";
        }
    }
}
```

6.4 LinkStatus

```
void LinkStatus(bool linkUp)
```

6.4.1 Remarks

This callback is called when the EPAS client library detects a change in the link status between the terminal and the ECR. When the *linkUp* parameter is true, the link is functioning. When the parameter is false then there has been a link failure and the ECR should take appropriate action.

6.4.2 Example initialisation

```
mEpas.LinkStatus = HandleLinkState;
```

6.4.3 Example usage

```
/// <summary>
/// Updates the ECR with link status notifications
/// </summary>
/// <param name="linkUp">When true, the link between the ECR and the terminal
/// is functioning.
/// When false, the link has failed.</param>
public void HandleLinkState(bool linkUp)
{
    if (InvokeRequired)
    {
        Invoke(new Action(() => HandleLinkState(linkUp)));
    }
    else
    {
        if (linkUp)
        {
            lblLink.BackColor = Color.PaleGreen;
            lblLink.ForeColor = Color.DarkGreen;
            lblLink.Text = "OK";
        }
        else
        {
            lblLink.BackColor = Color.LightPink;
            lblLink.ForeColor = Color.DarkRed;
            lblLink.Text = "DOWN";
        }
    }
}
```

6.5 RawMessageHandler

```
void RawMessageHandler(string rawXML)
```

6.5.1 Remarks

It may be useful or necessary for the ECR to be able to directly inspect EPAS XML messages that are received in the library. In these situations, the ECR can assign a handler to the *RawMessageHandler* callback.

This feature is only provided as a fallback way to deal with situations that the library does not handle. There is no clear usage for this, other than that it allows the ECR to directly handle EPAS messages to achieve some very specific objective that it cannot achieve via the library API.

In normal use the ECR should not need to provide a handler for this callback.

6.5.2 Example initialisation

```
mEpas.RawMessageHandler = HandleIncomingXML;
```

6.5.3 Example usage

```
/// <summary>
/// Handle raw EPAS XML messages as they arrive in the library
/// </summary>
/// <param name="rawXml">Raw EPAS XML message</param>
private void HandleIncomingXML(string rawXml)
{
    Log(LogLevel.Debug, "Incoming XML:");
    Log(LogLevel.Debug, rawXml);
}
```

7 Connecting to the terminal

After initialisation, the next step for the ECR is to connect to the terminal. The terminal supports connection by network or by USB. The choice is determined by configuration in the terminal; the terminal will only listen for a connection using the configured method.

Network connections are made using the *ConnectNetwork* method, while USB connections are made using the *ConnectUsb* method.

7.1 ConnectNetwork

```
ApiResult ConnectNetwork(TerminalNetworkConnection connectionDetails, bool forceReconnect
= false)
```

7.1.1 Remarks

The *ConnectNetwork* method attempts to connect to the terminal over a TCP/IP network connection. The *connectionDetails* parameter contains the details needed for the connection.

If the *forceReconnect* parameter is true then if there is already a connection in place that connection will be closed and a new one will be created. Otherwise, if *forceReconnect* is false, if there is already a connection in place then this method will just return ApiResult.OK.

7.1.2 The TerminalNetworkConnection class

The *TerminalNetworkConnection* class contains the following properties:

TerminalConnectionDetails

```
IPEndPoint TerminalConnectionDetails
```

The *TerminalConnectionDetails* property specifies the network address and port of the terminal. At time of writing the terminal only supports IPv4 connections.

Note: The default port for EPAS is 3000. This can be changed in the terminal, but there is usually no reason to do so.

UseKeepAlive

```
bool UseKeepAlive
```

If the *UseKeepAlive* method is true then the EPAS client library will ensure a steady stream of packets are sent to the terminal during idle times. This will allow the terminal to know that the ECR is still alive and connected.

Default value: false.

7.1.3 Example usage

```
/// <summary>
/// Connects to the terminal using the parameters set in the form fields.
/// </summary>
private void ConnectToTerminal()
{
    Log("Connecting...");

    bool dataOk = true;

    TerminalNetworkConnection tc = new TerminalNetworkConnection();
```

```
    try
    {
        tc.TerminalConnectionDetails = new
IPEndPoint(IPAddress.Parse(txtTerminalAddress.Text), int.Parse(txtTerminalPort.Text));
    }
    catch (Exception)
    {
        dataOk = false;
        LogError("Unable to parse terminal address and / or port");
    }

    tc.UseKeepAlive = optKeepalive.Checked;

    if (dataOk)
    {
        ApiResult result = mEpas.ConnectNetwork(tc, optForceReconnect.Checked);

        LogApiResult(result, "Connect");
    }
}
```

7.2 ConnectUsb

```
ApiResult ConnectUsb(string comPort, bool forceReconnect = false)
```

7.2.1 Remarks

The *ConnectUsb* method attempts to connect to the terminal over a USB connection.

When configured for USB communications, the terminal runs the USB connection in CDC mode. This causes the connected PC to create a virtual serial port. In Windows 10 this should not need any additional driver support.

The *comPort* parameter contains the name of this virtual serial port, e.g. “COM6”. The EPAS Client Library, however, will attempt to automatically detect the virtual serial port and if it can find a port that is connected to the terminal then it will use it. The value of *comPort* is only used if the library is unable to identify the port name automatically, in which case it will use *comPort* if it has been defined.

It is normally OK to leave the *comPort* value set to null or an empty string.

If the *forceReconnect* parameter is true then if there is already a connection in place that connection will be closed and a new one will be created. Otherwise, if *forceReconnect* is false, if there is already a connection in place then this method will just return ApiResult.OK.

7.2.2 Example usage

```
ApiResult result = mEpas.ConnectUsb(txtComPort.Text, optForceReconnect.Checked);
```

7.3 Disconnect

```
ApiResult Disconnect()
```

7.3.1 Remarks

The *Disconnect* method closes the connection to the terminal, whether it is network or USB.

7.3.2 Example usage

```
/// <summary>
/// Disconnect from the terminal
```

```
/// </summary>
private void btnDisconnect_Click()
{
    Log("Disconnecting");
    mEpas?.Disconnect();
}
```

8 Logging in

The ECR must log in to the terminal before it can run any transactions. The login includes necessary details about the terminal.

8.1 Login

```
ApiResult Login(TerminalEnvironment environment, out LoginResponse response)
```

8.1.1 Remarks

The Login method carries details of the operating environment for the terminal.

If the method returns ApiResult.OK then the *response* parameter contains the terminal's response to the login request.

8.1.2 The TerminalEnvironment class

TerminalId

```
string TerminalId
```

The Terminal ID is assigned by the terminal provider. The ID is used by the terminal to download configuration data, to identify itself in transactions, and to provide traceability in the case of transaction investigations.

This property must be set by the ECR.

ConfigurationServer

```
IPEndPoint ConfigurationServer
```

The configuration server is the server that the terminal must contact to download configuration data.

This property must be set by the ECR.

AuthorisationServer

```
IPEndPoint AuthorisationServer
```

The authorisation server is the server that the terminal will communicate with to authorise transactions. If this is left as null then the terminal will use the authorisation server details found in the downloaded configuration files.

In normal usage the ECR should set this property to identify the correct server details if possible.

OperatorId

```
string OperatorId
```

The operator ID identifies the cashier. This is not used by the terminal, but the value given is included in the receipt data for transactions.

ISO639_1_LanguageCode

```
string ISO639_1_LanguageCode
```

This property selects the language that the ECR texts should be translated to. The *TerminalEnvironment* class defines several language constants that can be used here:

- TerminalEnvironment.LangSwedish

- TerminalEnvironment.LangNorwegian
- TerminalEnvironment.LangDanish
- TerminalEnvironment.LangFinnish
- TerminalEnvironment.LangGerman
- TerminalEnvironment.LangEnglish
- TerminalEnvironment.LangPolish

Default value: TerminalEnvironment.LangSwedish

8.1.3 LoginResponse

The *LoginResponse* class contains the result of the login request.

LoggedIn

```
bool LoggedIn
```

If true, the login request was successful. Otherwise the login request failed and further details can be found in the *ErrorCondition* and *AdditionalInformation* properties.

ErrorCondition

```
ErrorConditions ErrorCondition
```

When a transaction fails the terminal should provide an indication of the reason for the failure in the *ErrorCondtition* property. Possible values are:

ErrorCondition value	Meaning
None	No further information was available.
Aborted	The Transaction was aborted.
Busy	The terminal was busy (see section 6.3).
Cancel	The transaction was cancelled on the terminal by the cardholder.
DeviceOut	A technical or internal error occurred in the terminal.
InProgress	A transaction is already being handled.
LoggedOut	The ECR is not logged in to the terminal.
MessageFormat	There was a fault in the format of an EPAS message received in the terminal.
NotAllowed	The message is not allowed at this time, e.g. a request was sent when a request was already being handled.
NotFound	A transaction could not be found, e.g. for a reversal.
Refusal	The transaction was declined.
UnavailableDevice	A device failure in the terminal meant the transaction could not be completed.
UnavailableService	The requested service is not available or not configured.
InvalidCard	The card presented by the customer is not valid in the terminal.
UnreachableHost	The transaction was required to go online but the authorisation host could not be reached / did not respond.
WrongPIN	The cardholder entered the wrong PIN.

AdditionalInformation

```
string AdditionalInformation
```

The terminal may provide additional diagnostic information in the *AdditionalInformation* property. This is intended for debug and logging purposes. The text in here is not translated and is not useful for a cashier or customer.

8.1.4 Example usage

```
TerminalEnvironment te = new TerminalEnvironment()
{
    AuthorisationServer = new IPEndPoint(IPAddress.Parse("185.27.171.42"), 55144),
    ConfigurationServer = new IPEndPoint(IPAddress.Parse("185.27.171.42"), 55133),
    ISO639_1_LanguageCode = TerminalEnvironment.LangSwedish,
    OperatorId = "cashier_1",
    TerminalId = "80000082"
};

if (epas.Login(te, out LoginResponse lr) == ApiResult.OK)
{
    if (lr.LoggedIn)
    {
        .
        .
        .
```

8.2 Logout

```
ApiResult Logout(out LogoutResponse response)
```

8.2.1 Remarks

The ECR should send a *Logout* request at the end of the day or when the cashier ends their shift. This ensures the terminal displays an appropriate message on screen.

If the method returns ApiResult.OK then the *response* parameter contains the terminal's response to the logout request.

8.2.2 LogoutResponse

The *LogoutResponse* class contains the result of the login request.

LoggedOut

```
bool LoggedOut
```

If true, the logout request was successful. Otherwise, the login request failed and further details can be found in the *ErrorCondition* and *AdditionalInformation* properties.

ErrorCondition
See section 0.

AdditionalInformation
See section 0.

8.2.3 Example usage

```
/// <summary>
/// Logs out of the terminal.
/// </summary>
```

```
private void LogoutOfTerminal()
{
    Log("Logging out...");

    LogoutResponse response;
    ApiResult result = mEpas.Logout(out response);
    LogApiResult(result, "Logout: " + response?.ToString());
}
```

9 Transactions

9.1 Introduction

The EPAS client library offers several transaction methods, and they are documented below.

Each transaction is synchronous, so the method will return when the transaction is complete. This can take some time, since it involves waiting for a card to be presented, waiting for PIN entry, etc.

Some of the methods in *IClientApp* will be called while the transaction is in progress. Similarly some of the optional handler methods will be called, if defined. All these methods will be called from a different thread to the one that is executing the transaction so there is no risk of deadlock in the callback mechanism.

9.1.1 Transaction types

The payment application supports the following transaction types:

- Purchase, which has three options:
  - Only the goods value is specified, and
  - All amount specified, i.e. goods value and cashback value,
  - No amounts specified.
- Cash Advance
- Refund
- Reversal

The functionality for these is detailed below.

9.2 Purchase transaction – goods value specified only

```
ApiResult Purchase(decimal purchaseAmount, out TransactionResponse response)
```

This is the simplest way to start a purchase transaction. The *purchaseAmount* parameter contains the goods value in the configured currency. There is no cashback in this method.

If the method returns ApiResult.OK then the result of the transaction is given in the *response* parameter.

9.2.1 Example usage

```
/// <summary>
/// Runs a simple purchase transaction for the given amount.  There is no cashback
/// or swipe-ahead in this transaction.
/// </summary>
/// <param name="amount">Transaction amount</param>
public void SimplePurchase(decimal amount)
{
    Log("Simple purchase of " + string.Format("{0:.00}", amount));

    TransactionResponse response;
    ApiResult result = mEpas.Purchase(amount, out response);
    if (result == ApiResult.OK)
    {
        HandleTxnResponse(response);
    }

    LogApiResult(result, "Simple purchase: " + response?.ToString());
}
```

9.3 Purchase with cashback

```
ApiResult Purchase(PurchaseTransactionAmounts transactionAmounts, out TransactionResponse response)
```

The *Purchase* method starts a purchase transaction with all the transaction amounts specified in the *transactionAmounts* parameter.

Note that if the cashback amount is zero then this is functionally identical to the method above.

If the method returns ApiResult.OK then the result of the transaction is given in the *response* parameter.

9.3.1 PurchaseTransactionAmounts

The PurchaseTransactionAmounts class contains the following properties:

PurchaseAmount

```
decimal PurchaseAmount
```

This is the value of the goods being purchased. It excludes any cashback amount.

CashbackAmount

```
decimal CashbackAmount
```

This is the value of cashback that is requested. This is added to *PurchaseAmount* to produce the requested transaction value.

9.3.2 Example usage

```
/// <summary>
/// Runs a full-featured purchase transaction
/// </summary>
/// <param name="amount">Purchase amount</param>
/// <param name="cashback">Cashback amount</param>
public void PurchaseWithCashback(decimal amount, decimal cashback)
{
    Log("Purchase of " + string.Format("{0:.00}", amount) +
        " + " + string.Format("{0:.00}", cashback) +
        " cashback");

    TransactionResponse response;

    PurchaseTransactionAmounts txnAmount = new PurchaseTransactionAmounts(amount, cashback);

    ApiResult result = mEpas.Purchase(txnAmount, out response);
    if (result == ApiResult.OK)
    {
        HandleTxnResponse(response);
    }

    LogApiResult(result, "Purchase: " + response?.ToString());
}
```

9.4 Purchase with no amounts defined

```
ApiResult PurchaseBeforeAmount(out TransactionResponse response)
```

This method starts a purchase transaction before the amounts are known. The cardholder will be able to present a card and enter a PIN, and the terminal will then wait for the amounts to be supplied via the *SendPurchaseAmounts* method. The terminal will then prompt the cardholder to accept the amount.

Note that *SendPurchaseAmounts* can be sent any time after the transaction is started. It does not have to be sent at a specific stage in the transaction sequence.

If the method returns ApiResult.OK then the result of the transaction is given in the *response* parameter.

9.4.1 Example usage

```
/// <summary>
/// Runs a pre-amount purchase transaction, where the amount is supplied later.
/// Also known as swipe-ahead and pre-tap.
/// </summary>
public void PreAmountPurchase()
{
    Log("PreAmount purchase");

    TransactionResponse response;
    ApiResult result = mEpas.PurchaseBeforeAmount(out response);
    if (result == ApiResult.OK)
    {
        HandleTxnResponse(response);
    }

    LogApiResult(result, "PreAmount purchase: " + response?.ToString());
}
```

9.5 Supplying purchase amounts after the transaction has been started

```
ApiResult SendPurchaseAmounts(PurchaseTransactionAmounts transactionAmount)
```

When *PurchaseBeforeAmount* is used then the transaction amounts are provided using the *SendPurchaseAmounts* method.

The *transactionAmount* parameter is an instance of the *PurchaseTransactionAmounts* class, defined in section 9.3.1.

9.5.1 Example usage

```
/// <summary>
/// Send the transaction amounts for a pre-amount purchase transaction.
/// </summary>
/// <param name="amount">Purchase amount</param>
/// <param name="cashback">Cashback amount</param>
public void SendAmounts(decimal amount, decimal cashback)
{
    Log("Sending amounts: " + string.Format("{0:.00}", amount) +
        " + " + string.Format("{0:.00}", cashback) +
        " cashback");
    PurchaseTransactionAmounts txnAmount = new PurchaseTransactionAmounts(amount, cashback);
    LogApiResult(mEpas.SendPurchaseAmounts(txnAmount), "SendPurchaseAmounts");
}
```

9.6 Cash advance transaction

```
ApiResult CashAdvance(decimal cashAmount, out TransactionResponse response)
```

This method starts a cash advance transaction for the specified *cashAmount* value.

If the method returns ApiResult.OK then the result of the transaction is given in the *response* parameter.

Note that the terminal must be configured to allow cash advance transactions for this to be successful.

9.6.1 Cash advance charges

The terminal may be configured to apply a transaction fee for cash advance transactions. This fee is included in the final transaction total and is specified in the receipt data for the transaction (see section 3.5).

9.6.2 Example usage

```
/// <summary>
/// Runs a cash advance transaction.
/// </summary>
/// <param name="amount">Cash amount</param>
/// sending the transaction result</param>
public void CashAdvance(decimal amount)
{
    Log("Cash Advance of " + string.Format("{0:.00}", amount));

    TransactionResponse response;

    ApiResult result = mEpas.CashAdvance(amount, out response);
    if (result == ApiResult.OK)
    {
        HandleTxnResponse(response);
    }

    LogApiResult(result, "Cash Advance: " + response?.ToString());
}
```

9.7 Card payment refund transaction

```
ApiResult Refund(decimal refundAmount, out TransactionResponse response)
```

This method starts a refund transaction of a card purchase for the specified *refundAmount* value.

If the method returns ApiResult.OK then the result of the transaction is given in the *response* parameter.

Note that the receipt layout and contents for a refund transaction are different to those of a purchase transaction (see section 3.5).

9.7.1 Example usage

```
/// <summary>
/// Runs a refund transaction
/// </summary>
/// <param name="amount">Refund amount</param>
/// sending the transaction result</param>
public void Refund(decimal amount)
{
    Log("Refund of " + string.Format("{0:.00}", amount));

    TransactionResponse response;

    ApiResult result = mEpas.Refund(amount, out response);
    if (result == ApiResult.OK)
    {
        HandleTxnResponse(response);
    }

    LogApiResult(result, "Refund: " + response?.ToString());
}
```

9.8 Alternative payment method refund (not yet implemented)

```
ApiResult Refund(AltMethodTransaction transactionDetails, out TransactionResponse
response)
```

This method starts a refund of a transaction made using an alternative payment method, which means a payment method that did not involve using a card with the terminal.

Note: This method is here as a placeholder for integration and is not implemented in v1.0 of the client library.

Currently the only alternative payment method supported is Swish. Swish payment are handled transparently on the terminal, but a refund requires a specific indication of the method involved.

9.8.1 AltMethodTransaction

The AltMethodTransaction class is an abstract class. It is used here to facilitate future expansion. The library provides derived classes to handle different alternative payment methods. Currently the only implementation of this is the SwishTransaction class.

9.8.2 SwishTransaction

The SwishTransaction class contains information relating to a Swish transaction that is needed for a refund to be made. The values are set by the constructor.

```
SwishTransaction(decimal amount, string transactionReference, string phoneNumber)
```

The *amount* parameter is the amount of the Swish transaction to be refunded. Swish supports partial refunds, so this can be less than the full transaction amount.

The *reference* parameter is the Swish transaction reference.

The *phoneNumber* parameter is the customer's phone number for the phone that made the original purchase transaction.

9.8.3 Example usage

```
SwishTransaction txn = new SwishTransaction(499, "1234567890AABBCCAABBCC", "0728315570");
if (epas.Refund(txn, out TransactionResponse refundResponse) == ApiResult.OK)
{
    // Process the refund response as normal
}
```

9.9 Reversal transaction

```
ApiResult ReverseLastTransaction(string transactionReference, out TransactionResponse
response)
```

A reversal is a request to cancel the most recent transaction. A transaction can only be reversed if it was approved.

The *transactionReference* parameter is the reference that was provided in the response of the transaction to be reversed, as detailed in section 9.10.2.

9.9.1 Example usage

```
/// <summary>
/// Reverse the last transaction
/// </summary>
public void Reversal()
```

```
{
    Log("Reversal");

    TransactionResponse response;

    ApiResult result = mEpas.ReverseLastTransaction(txnRef.Text, out response);
    if (result == ApiResult.OK)
    {
        HandleTxnResponse(response);
    }

    LogApiResult(result, "Reversal: " + response?.ToString());
}
```

9.10 TransactionResponse

All transactions, if executed correctly, will provide a response in an instance of the *TransactionResponse* class.

9.10.1 Approved

```
bool Approved
```

If this is true, then the transaction was approved. Otherwise, the *ErrorCondition* and *AdditionalInformation* properties will give more detail on why the transaction was not approved.

9.10.2 TransactionReference

```
string TransactionReference
```

The transaction reference is assigned by the terminal. It identifies the transaction and is required for a reversal, as well as for any post-transaction checks or queries that might arise.

9.10.3 FinalAmount

```
decimal FinalAmount
```

The final value of the transaction will be the sum of *PurchaseAmount*, *CashbackAmount* and any tip (see section 0) or charges (see section 0) that are added.

The transaction amount may also be affected by DCC (see section 3.6).

The breakdown of the final transaction amount is given in the receipt data (see section 3.5).

9.10.4 ErrorCondition

See section 0.

9.10.5 AdditionalInformation

See section 0.

9.11 Aborting a transaction

```
ApiResult RequestTransactionAbort()
```

All transactions methods are synchronous, i.e. the method will return when the transaction is complete. There may be situations where the transaction must be cancelled, e.g. if the customer changes their mind and walks away.

To do this, the ECR can call the *RequestTransactionAbort* method from a different thread (since the transaction thread is blocked at this stage until a response arrives). This method will send an abort request to the terminal.

There is no response to the abort request. Instead, the transaction will be cancelled, if possible, and the transaction response will be sent back when the terminal has processed the abort request and has completed the transaction.

Note that it is possible for the abort request to be received in the terminal at a point where it is too late to be processed. The ECR should, therefore, always correctly process the transaction result and the receipts, and should not assume that the transaction will definitely be aborted just because the request was sent.

10 Card handling

The EPAS Client Library offers two methods relating to card handling.

10.1 EnableCardReaders

```
ApiResult EnableCardReaders(bool enable, out EnableReadersResponse response)
```

10.1.1 Remarks

The *EnableCardReaders* requests will, if successful, enable or disable the terminal's chip and magnetic card readers before a transaction is launched. If the *enable* parameter is true then the request is to enable the card readers. If false, the request is to disable the card readers.

If the method returns ApiResult.OK then the *response* parameter contains the terminal's response to the enable card readers request.

When the card readers are enabled then cards will be processed in the same way as if there was a transaction in progress, i.e. the ECR will get card notifications and loyalty card processing (see section 5.8) will operate as normal.

The terminal will go as far as it can through the normal processes until it needs to know what transaction is involved, at which point it will wait for a transaction to be requested.

The ECR can, therefore, use the *EnableCardReaders* method to do loyalty card functions such as registration, checking a balance, etc. Then once the card is processed it would call *EnableCardReaders* with the *enable* parameters set to false to close the card readers again. The CardStatus (see section 6.1) events will allow the ECR to prompt for card removal if desired.

10.1.2 EnableReadersResponse

The *EnableReadersResponse* class contains the result of the enable card readers operation request.

Successful

```
bool Successful
```

If true, the request to enable / disable the card readers was successful. Otherwise, the request failed and further details can be found in the *ErrorCondition* and *AdditionalInformation* properties.

ErrorCondition
See section 0.

AdditionalInformation
See section 0.

10.1.3 Example usage

```
/// <summary>
/// Enable or disable the card readers
/// </summary>
/// <param name="enable">True to enable, false to disable</param>
public void HandleEnableReaders(bool enable)
{
    Log((enable ? "Enabling" : "Disabling") + " card readers");
    EnableReadersResponse response;
    ApiResult result = mEpas.EnableCardReaders(enable, out response);
    LogApiResult(result, (enable ? "Enable" : "Disable") + " reader : " +
        response?.ToString());
```

```
}
```

11 Additional functionality

The functionality described here is not specifically related to transactions.

11.1 RequestAdministrativeOperation

```
ApiResult RequestAdministrativeOperation(AdminOperation operation, out AdminResponse response)
```

11.1.1 Remarks

An administrative operation is one that has a very specific purpose in the terminal.

In all cases, if the request returns ApiResult.OK then the response from the terminal is found in the *response* parameter.

The operations current supported are:

AdminOperation.HostLogin
This operation requests the terminal to run a login transaction with the authorisation host. The terminal will do this automatically when needed, but there are times where it can be useful to explicitly perform the login, e.g. to check that the authorisation host is reachable and functioning.

The host login also will attempt to upload any transactions that are on the Store & Forward queue on the terminal, so it may be that the ECR chooses to do this at the end of the day to make sure all transactions are committed.

AdminOperation.UpdateParameters
The terminal will normally get an indication that new parameters are ready to be downloaded and installed when it completes an online transaction, either a financial transaction or a login. But the ECR can use this operation to force the terminal to go online, check with the configuration server, and download any new configuration data that is available.

Note that a successful result here does not indicate that new parameters were downloaded. If the update check showed that there were no new parameters available then the operation result will still be successful.

11.1.2 AdminResponse

The *AdminResponse* class contains the result of the administrative operation request.

Successful

```
bool Successful
```

If true, the administrative operation request was successful. Otherwise, the request failed and further details can be found in the *ErrorCondition* and *AdditionalInformation* properties.

ErrorCondition
See section 0.

AdditionalInformation
See section 0.

11.1.3 Example usage

```
/// <summary>
/// Download any configuration updates that are available
/// </summary>
```

```
public void DoConfigUpdate()
{
    AdminResponse response;

    ApiResult result = mEpas.RequestAdministrativeOperation(AdminOperation.UpdateParameters,
out response);

    LogApiResult(result, "Configuration update: " + response?.ToString());
}
```

11.2 SendRawXml

```
ApiResult SendRawXml(string xml)
```

11.2.1 Remarks

Section 6.5 describes how the ECR has the option, if required, to handle raw EPAS XML message as they are received from the terminal. This method is the counterpart to that functionality, and it provides the ECR with a way to send a raw EPAS XML message to the terminal.

There is no value in describing how to use this here, since it requires a thorough knowledge of the EPAS protocol. This is very much for advanced usage only.

11.2.2 Example usage

```
/// <summary>
/// Send a block of raw EPAS XML to the terminal
/// </summary>
private void btnSendXml()
{
    string xml = txtRawXML.Text;
    if (!string.IsNullOrEmpty(xml))
    {
        LogApiResult(mEpas.SendRawXml(xml), "Send raw XML");
    }
}
```

12 Receipt data and methods

The biggest challenge in receipt handling is in dealing with all the different receipt rules, layouts and contents that may occur. It is strongly recommended to study the Cardholder and Operator Interface specification (document ref 2). This document can be provided by Westpay. Section 8 of that document is dedicated to receipt layouts and contents and it is worth becoming familiar with the receipt layouts and the different factors that are involved.

Note, however, that the CHAOI specification is quite old now, and does not contain information relating to contactless card use or off-terminal transaction flows.

This section of the API specification aims to explain what data is provided and what it means. The construction of receipts, however, is beyond the scope of this document.

12.1 ReceiptData

The *ReceiptData* class is provided by the EPAS Client Library, and it contains all the data required to generate a transaction receipt. Most of the data is contained in other classes, and these are documented below.

12.1.1 Version

```
ReceiptVersion Version
```

Indicates if the receipt is a customer or a merchant receipt.

ReceiptVersion value	Meaning
Merchant	The receipt is intended for the merchant.
Cardholder	The receipt is intended for the cardholder.

12.1.2 Outcome

```
TransactionOutcome Outcome
```

Indicates the result of the transaction.

TransactionOutcome value	Meaning
Approved	The transaction was approved.
Declined	The transaction was declined, either by the authorisation host or by the terminal.
Cancelled	The transaction was cancelled, either by the cardholder or by the ECR.

12.1.3 AmountFormatSpecifier

```
string AmountFormatSpecifier
```

The *ReceiptData* class provides helper functions that can involve formatting amounts as text. The *AmountFormatSpecifier* string indicates how these amounts should be formatted and is defined in https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-numeric-format-strings.

The default value of *AmountFormatSpecifier* is “F2”, which indicates amounts should be presented as floating-point values with two decimal places.

12.1.4 TerminalID

```
string TerminalID
```

Contains the terminal ID, which is provided originally by the ECR in the terminal's operating parameters (see section 0). Used in the receipt header.

12.1.5 OperatorID

```
string OperatorID
```

Contains the cashier / operator ID, which is provided originally by the ECR in the terminal's operating parameters (see section 0). Used in the receipt header.

12.1.6 BankAgent

```
string BankAgent
```

Contains the name of the bank agent. Used in the receipt header.

12.1.7 AcquirerReference

```
string AcquirerReference
```

Contains the acquirer reference number. Used in the receipt header.

12.1.8 Transaction

```
TransactionData Transaction
```

Contains the details of the financial transaction. See section 0.

12.1.9 MerchantDetails

```
MerchantData MerchantDetails
```

Contains details relating to the merchant. See section 12.3.

12.1.10 AuthorisationData

```
AuthorisationData Authorisation
```

Contains data relating to the transaction authorisation. See section 12.4.

12.1.11 CardData

```
CardData Card
```

Contains details of the card used in the transaction. See section 12.5.

12.1.12 EmvData

```
EmvData EMV
```

Contains EMV data elements if an EMV card was used in the transaction. See section 12.6.

12.1.13 DccData

```
DccData DCC
```

Contains the DCC data, if chosen by the customer. See section 12.7.

12.1.14 CodeString

```
string CodeString()
```

This helper method generates a code string based on the transaction data. The code string is defined in CHAOI (document ref 2) in section 8.3.1, where the example given is “C@1 5 000 SVB 012 12233”. The meaning of each of these elements is discussed in section 8 of the CHAOI specification.

12.1.15 CardholderLanguage

```
string CardholderLanguage
```

Contains the two-letter ISO 639-1 language code that indicates the cardholder language used on the terminal. The ECR may use this to add localised text to the receipt.

12.1.16 GetTranslatedText

```
string GetTranslatedText(TranslatedTextId textId, string suffix = null)
```

This helper method provides translations of standard receipt texts. The text is identified by the *textId* parameter.

TranslatedTextId value	Meaning
Outcome	The transaction outcome, i.e. approved, declined or cancelled.
DenialReason	If the transaction was declined this text provides added information on the reason
TransactionType	Transaction type, e.g. "purchase", "refund", etc.
PinUsed	Text that indicates a PIN was used
Sign	Text that indicates the receipt should be signed by the customer
Contactless	Text that indicates a contactless transaction
VerifiedByDevice	Text that indicates the customer identity was verified by a consumer device, e.g. a mobile phone
ReferenceNumber	Row header for the transaction reference
OrganisationNumber	Row header for the organisation number
CashierId	Row header for the cashier / operator ID
CashierSignature	Prompt for the cashier's signature in a refund transaction
CashierName	Prompt for the cashier's name in a refund transaction
Total	Row header for total amount of the transaction
VATAmount	Row header for the VAT amount
CashbackAmount	Row header for the cashback amount
AcquirerFee	Additional acquirer fee

AcquirerReferenceNumber	Row header for the acquirer reference number
TerminalId	Row header for the terminal ID
RecipientTag	Typically "cardholder copy" or "merchant copy", to show the merchant which receipt is being printed
PaymentCode	Payment code, may be required by the payment card and is provided by the ECR
ApprovalMessage	Text such as AUTHORISATION FOR CREDITING ABOVE ACCOUNT that applies to signature bars
DebitOrCredit	Some older cards require the cardholder to choose between their debit or credit account. If that choice is made then the appropriate translation will be provided. If the choice did not apply then no translation is provided.

The *suffix* parameter is appended to the translated text, but only if the text is not blank.

12.1.17 GenerateSimpleReceipt

```
string GenerateSimpleReceipt(int receiptFixedWidthChars)
```

The GenerateSimpleReceipt method provides a quick way to produce a formatted receipt.

The *receiptFixedWidthChars* parameter indicates the width in characters of the receipt. This has a minimum value of 24 characters.

Note: The GenerateSimpleReceipt method is intended only for development and debug purposes. The requirements on receipt contents change according to location, environment, and issuer / processor requirements. Developers using the EPAS Client Library should implement the receipt layout that is required for the situation in which the library will be used.

12.2 ReceiptData.TransactionData

The *TransactionData* class contains transaction details such as the transaction type, time, and amounts.

12.2.1 TxnType

```
TransactionType TxnType
```

Indicates the type of transaction that was performed.

TransactionType value	Meaning
Unknown	This should never be seen, but it is possible that the terminal software could be updated with a new transaction type that is not recognised by the EPAS Client Library, in which case the transaction type will not be recognised and the Unknown value will be used.
Purchase	Purchase transaction, with or without cashback.
Refund	Refund transaction.
CashAdvance	Cash advance transaction.
PurchaseReversal	Reversal of a purchase transaction.
RefundReversal	Reversal of a refund transaction.

12.2.2 Timestamp

```
DateTime Timestamp
```

The date and time of the transaction. Used in the receipt header.

12.2.3 Reference

```
string Reference
```

The transaction's reference. Used in the receipt header.

12.2.4 IsoCurrencyCode

```
string IsoCurrencyCode
```

Contains the three-letter ISO currency code used in the transaction, e.g. "SEK", "EUR", etc. This code is derived from the transaction currency code that is set in the operating parameters (see section 0). The code can be used without modification on the receipt, or the ECR can work out formatting parameters based on the currency selection.

12.2.5 AlternativePaymentMethod

```
string AlternativePaymentMethod
```

Contains the name of any alternative payment method that was used. An alternative payment method is where the normal card process was not used, and instead the customer paid by other means. At time of writing the only supported alternative payment method is Swish, in which case the value here will be "Swish".

12.2.6 NetAmount

```
decimal NetAmount
```

The value of the transaction, without cashback, tip, or charges included. The VAT value, below, is part of the net amount.

12.2.7 Cashback

```
decimal Cashback
```

The cashback component of the transaction.

12.2.8 Vat

```
decimal Vat
```

The VAT component of the net amount, if specified.

12.2.9 Tip

```
decimal Tip
```

The value of the tip, if added by the customer.

12.2.10 AdditionalCharges

```
decimal AdditionalCharges
```

The value of additional charges. These normally only apply in cash advance transactions.

12.2.11 Total

```
decimal Total
```

The total of the net amount, cashback amount, tip, and additional charges.

12.3 ReceiptData.MerchantData

The *MerchantData* class contains details of the merchant. These are taken from the terminal's configuration data, and it is reasonable for the ECR to ignore these values and use merchant details that are configured directly in the ECR.

12.3.1 Name

```
string Name
```

The name of the merchant.

12.3.2 Address

```
string Address
```

The merchant's street address.

12.3.3 City

```
string City
```

The merchant's city.

12.3.4 Zip

```
string Zip
```

The merchant's zip / postal code.

12.3.5 PhoneNumber

```
string PhoneNumber
```

The merchant's phone number

12.3.6 OrganisationNumber

```
string OrganisationNumber
```

The merchant's organisation number.

12.3.7 MerchantSummary

```
string MerchantSummary()
```

Helper method that provides a multi-line string that includes the defined parts of the merchant details.

12.4 ReceiptData.AuthorisationData

The *AuthorisationData* class contains details of the authorisation of the transaction.

12.4.1 CardholderVerificationMethod

```
VerificationMethod CardholderVerificationMethod
```

Indicates how the cardholder's identity was verified.

CardholderVerificationMethod value	Meaning
Signature	The cardholder identity was verified by signature.
PINOnline	The cardholder identity was verified by online PIN.
PINOffline	The cardholder identity was verified by offline PIN.
NoVerification	The cardholder identity was not verified.
NoCVM	The low-value 'No CVM' rule applied in this transaction.
CVMFailed	Cardholder identity verification failed.
Combined	The cardholder identity was verified by both signature and offline PIN.
ConsumerDevice	The cardholder identity was verified by a consumer device, typically a mobile phone.

Note that there are specific receipt guidelines around the *ConsumerDevice* verification method. Swedbank (document ref 3) says that if the cardholder verification is done in the device then the receipt should indicate that offline PIN was used.

The character enumeration values for each of these corresponds to the associated code in CHAOI (document ref 2) section 8.1.3.

12.4.2 Channel

```
AuthorisationChannel Channel
```

Indicates what method was used to authorise the transaction.

AuthorisationChannel value	Meaning
Online	Transaction was authorised online.
Offline	Transaction was authorised offline.
Phone	Transaction was authorised by phone call.
None	Transaction was not authorised.
DeclinedBeforeAuth	Transaction was declined before authorisation was attempted.

The authorising entity is documented in CHAOI (document ref 2) section 8.1.4. The character enumeration value of each on corresponds to the CHAOI specification.

12.4.3 Responder

```
AuthorisingEntity Responder
```

Indicates which entity authorised the transaction.

AuthorisingEntity value	Meaning
DPC	Authorised by the data processing centre.
LocalDevice	Authorised locally in the terminal.
DPC3	Authorised by the data processing centre.
DPC4	Authorised by the data processing centre.
CardIssuer	Authorised by the card issuer.
InterchangeInterface	Interchange interface process.
InterchangeVisaEpss	Interchange (VISA or EPSS).
Merchant	Authorised by merchant (acquiring) host.
ManualEntry	Approval code was manually entered.
DeclinedBeforeAuth	Transaction was declined before authorisation was attempted.

The authorising entity is documented in CHAOI (document ref 2) section 8.1.5. The character enumeration value of each on corresponds to the CHAOI specification.

12.4.4 ResponseCode

```
string ResponseCode
```

Contains the authorisation response code. This is the code that was returned from the authorisation process to indicate the result of the authorisation. The code is documented in CHAOI (document ref 2) section 8.1.6.

12.4.5 FinancialInstitution

```
string FinancialInstitution
```

The three-letter financial institution code.

12.4.6 ApprovalCode

```
string ApprovalCode
```

If the transaction was approved, this is the approval code issued by the authorising entity.

12.4.7 VerifiedByDevice

```
bool VerifiedByDevice
```

Indicates that the cardholder verification was performed in a separate device. Examples of these are ApplePay, GooglePay, etc., where the cardholder verification does not involve the terminal.

12.4.8 PinUsed

```
bool PinUsed()
```

This helper method will return true if the cardholder verification method used a PIN in some way. The receipt guidelines indicate that "PIN USED" should be included in the receipt in these cases.

12.4.9 SignatureRequired

```
bool SignatureRequired()
```

This helper method will return true if the cardholder verification method uses signature. In these cases the merchant receipt should include a space for the signature to be written.

12.5 ReceiptData.CardData

The *CardData* class contains details of the card used in the transaction.

12.5.1 MaskedCardNumber

```
string MaskedCardNumber
```

Card numbers are never supplied in plain text.

- In cardholder receipts the card number is masked to only show the last four digits.
- In merchant receipts the card number is masked to show the first six and last four digits.

12.5.2 CardName

```
string CardName
```

The name of the card, e.g. "Visa", "MasterCard", etc.

12.5.3 CardPresentation

```
EntryMethod CardPresentation
```

EntryMethod value	Meaning
ChipInserted	The card was inserted to the chip reader.
Swiped	The card was swiped.
Manual	The card number was entered manually.
Contactless	The card was tapped, or an off-terminal method was used.
Unknown	Used if the method from the terminal is not recognised by the library.

The card entry methods are documented in CHAOI (document ref 2) section 8.1.2. The character enumeration value of each on corresponds to the CHAOI specification. Note that CHAIO does not recognise contactless transactions. The entry method code for contactless transaction is 'K'.

Note that contactless card transactions should include the text "CONTACTLESS" on the receipt. This can be in English or translated to the appropriate language.

12.5.4 PaymentCode

```
string PaymentCode
```

The payment code is the code that was provided by the ECR. See section 5.6 for details.

12.6 ReceiptData.EmvData

The *EmvData* class contains EMV data elements for transactions where EMV was involved.

12.6.1 ApplicationId

```
string ApplicationId
```

The EMV application ID (AID) identifies the chip application that was used in the transaction. This should be included in the receipt.

12.6.2 TerminalVerificationResults

```
string TerminalVerificationResults
```

The terminal verification results (TVR) property is a sequence of ten hex digits that provide verification information about an EMV transaction. This should be included in the receipt.

12.6.3 TransactionStatusIndicator

```
string TransactionStatusIndicator
```

The transaction status indicator (TSI) is a sequence of four hex digits that provide status information about an EMV transaction. This should be included in the receipt.

12.6.4 LongReceiptText

```
string LongReceiptText()
```

This helper method creates a three-line text block containing the AID, TVR and TSI on separate lines.

12.6.5 CompactReceiptText

```
string CompactReceiptText()
```

This helper method creates a two-line text block containing the AID on the first line and the TVR and TSI on the second line.

12.7 ReceiptData.DccData

The *DccData* class is present where a transaction used DCC (see section 3.6).

The ECR can use the pre-set disclaimer text in the *Disclaimer* property, or it can use the various elements to create a disclaimer text.

12.7.1 Provider

```
string Provider
```

The DCC rate provider.

12.7.2 Source

```
string Source
```

The DCC rate source.

12.7.3 Rate

```
decimal Rate
```

The conversion rate applied to the transaction value.

12.7.4 Amount

```
decimal Amount
```

The converted total amount of the transaction in the cardholder's own currency.

12.7.5 RateTimestamp

```
DateTime RateTimestamp
```

The timestamp when the conversion rate was provided.

12.7.6 IsoCurrencyNumber

```
int IsoCurrencyNumber
```

The ISO 4217 currency number of the converted currency, i.e. the currency of the card.

12.7.7 IsoCurrencyCode

```
string IsoCurrencyCode
```

The ISO 4217 currency code of the converted currency, i.e. the currency of the card.

12.7.8 Exponent

```
int Exponent
```

The number of digits that follow the decimal point in the converted currency. Nearly all currencies have two digits following the point, but this should not be assumed.

12.7.9 MarkUp

```
decimal MarkUp
```

The conversion charge / mark up as a percentage. This is included in the total amount.

12.7.10 MarkUpDescription

```
string MarkUpDescription
```

If this property has a value then the text should be printed on the receipt after the mark up / disclaimer is printed.

12.7.11 Tip

```
decimal Tip
```

The value of any added tip, given in the converted currency.

12.7.12 **CardScheme**

```
string CardScheme
```

Indicates the card scheme used in the DCC transaction. The value here is based on configuration data in the terminal, and there is scope for the range of values to expand over time, therefore this property is a string rather than an enumeration.

At time of writing, possible values are:

- "V": A Visa card was used.
- "M": A MasterCard card was used.

This is significant because the disclaimer text changes depending on the card scheme. The disclaimer text also changes depending on the transaction type and the DCC provider.

If the ECR needs to use a different disclaimer text then it must take these different factors into account.

12.7.13 **Disclaimer**

```
string Disclaimer
```

Contains the disclaimer text that must be included in receipts for transactions that use DCC. The DCC text is always presented in English. The text contains the required elements, but does not include the *MarkUpDescription* property, which must be printed if provided.

13 Transaction flows

This section shows some procedural and transaction flow diagrams to help understand a typical sequence of callbacks during a transaction.

13.1 Connection

Client		Terminal
ConnectNetwork	→	
	←	ConnectNetwork: returns OK
	←	callback: LinkStatus(True)
Login	→	
	←	callback: BusyStatus(True)
	←	callback: Display("Please wait")
	←	callback: BusyStatus(False)
	←	Login: returns OK
	←	callback: Display("Welcome")
Transactions happen, etc.		
Disconnect	→	
	←	callback: LinkStatus(False)

13.2 Simple transaction

This is a simple transaction example. The client application calls the Purchase method and then receives several callbacks, but does not have to provide any further information to the terminal.

Client		Terminal
Purchase	→	
	←	callback: Display("New customer / Waiting for card")
	←	callback: CardStatus(True)
	←	callback: Display("Please wait")
	←	callback: Display("Please wait")
	←	callback: CardAccepted("541333******0020")
	←	callback: Display("MasterCard / Ask for PIN / Signature possible")
	←	callback: Display("Please wait")
	←	callback: Display("Please wait")
	←	callback: PrintReceipt, version=Merchant
	←	callback: PrintReceipt, version=Cardholder
	←	callback: Display("PURCHASE / Approved / Ask customer to take card")

	←	callback: CardStatus(False)
	←	callback: Display("Approved")
	←	Purchase returns OK
	←	callback: Display("Welcome")

13.3 Pre-amount transaction

This sequence shows a pre-amount transaction, where the transaction is started with no amounts specified, and the amounts are sent later.

Client		Terminal
PurchaseBeforeAmount	→	
	←	callback: Display("New customer / Waiting for card")
	←	callback: CardStatus(True)
	←	callback: Display("Please wait")
	←	callback: CardAccepted("541333******0020")
	←	callback: Display("MasterCard / Ask for PIN / Signature possible")
	←	callback: Display("Please wait")
	←	callback: Display("Chip is read / Waiting for amount")
SendPurchaseAmounts	→	
	←	SendPurchaseAmounts response OK
	←	callback: Display("Confirm total amount / MasterCard / 550,00 SEK")
	←	callback: Display("Please wait")
	←	callback: PrintReceipt, version=Merchant
	←	callback: Display("Purchase approved / Ask customer to take card")
	←	callback: CardStatus(False)
	←	callback: Display("Purchase approved")
	←	PurchaseBeforeAmount response is OK
	←	callback: Display("Welcome")

In this example the amounts are sent after the “Waiting for amount” display prompt, but that is not a requirement. Here is a sequence that shows the amount being sent before the card is inserted. Note that there is no “confirm total amount” display, because the amount is shown on the PIN entry screen so the customer has the option to cancel the transaction if the amount is wrong.

Client		Terminal
PurchaseBeforeAmount	→	
	←	PurchaseBeforeAmount
	←	callback: Display("New customer / Waiting for card")

SendPurchaseAmounts	→	
	←	SendPurchaseAmounts response OK
	←	callback: Display("New customer / Waiting for card")
	←	callback: CardStatus(True)
	←	callback: Display("Please wait")
	←	callback: CardAccepted("541333******0020")
	←	callback: Display("MasterCard / Ask for PIN / Signature possible")
	←	callback: Display("Please wait")
	←	callback: PrintReceipt, version=Merchant
	←	callback: Display("Purchase approved / Ask customer to take card")
	←	callback: CardStatus(False)
	←	callback: Display("Purchase approved")
	←	PurchaseBeforeAmount response is OK
	←	callback: Display("Welcome")

13.4 Purchase with signature verification

The sequence below shows the terminal asking for signature verification during a purchase, where the cardholder has chosen signature rather than PIN. Note that the merchant receipt is printed earlier than in a non-signature transaction, because the merchant receipt is the one that is signed by the customer.

Note also that the card is removed before signature verification in this sequence.

Client		Terminal
Purchase	→	
	←	callback: Display("New customer / Waiting for card")
	←	callback: CardStatus(True)
	←	callback: Display("Please wait")
	←	callback: CardAccepted("541333******0020")
	←	callback: Display("MasterCard / Ask for PIN / Signature possible")
	←	callback: Display("Please wait")
	←	callback: PrintReceipt, version=Merchant
	←	callback: Display("550,00 SEK / MasterCard / Ask customer to take card / Verify signature")
	←	callback: CardStatus(False)
	←	callback: VerifySignature
VerifySignature returns True, signatureOk=True	→	
	←	callback: PrintReceipt, version=Cardholder

	←	callback: Display("Purchase approved")
	←	Purchase response is OK

13.5 Cancelled transaction

A cancelled transaction is where the cardholder chooses to end the transaction early. In this sequence the card is removed at the PIN entry stage. Note the CardStatus callback showing the card removal. Note also that only one receipt is printed here because there is normally no merchant receipt for cancelled transactions.

Client		Terminal
Purchase	→	
	←	callback: Display("New customer / Waiting for card")
	←	callback: CardStatus(True)
	←	callback: Display("Please wait")
	←	callback: CardAccepted("541333******0020")
	←	callback: Display("MasterCard / Ask for PIN / Signature possible")
	←	callback: CardStatus(False)
	←	callback: Display("Transaction aborted")
	←	callback: PrintReceipt, version=Cardholder
	←	Purchase response is OK

13.6 Aborted transaction

An aborted transaction is where the transaction is ended early by the client application. As detailed in the API documentation, there is no response to the RequestTransactionAbort method. The client application must wait for the transaction result to be returned and then act accordingly.

Client		Terminal
Purchase	→	
	←	callback: Display("New customer / Waiting for card")
	←	callback: CardStatus(True)
	←	callback: Display("Please wait")
	←	callback: CardAccepted("541333******0020")
	←	callback: Display("MasterCard / Ask for PIN / Signature possible")
RequestTransactionAbort	→	
	←	callback: Display("Transaction aborted / Ask customer to take card")
	←	callback: PrintReceipt, version=Cardholder
	←	callback: CardStatus(False)

	←	callback: Display("Transaction aborted")
	←	Purchase response is OK

13.7 Refund transaction

A refund transaction behaves in the same way as a simple purchase transaction. It is simpler from the customer's perspective because there is no PIN entry. The authorisation and authentication responsibility in a refund is on the merchant rather than the card issuer.

Client		Terminal
Refund	→	
	←	callback: Display("New customer / Waiting for card")
	←	callback: CardStatus(True)
	←	callback: Display("Please wait")
	←	callback: CardAccepted("541333******0020")
	←	callback: Display("Please wait")
	←	callback: PrintReceipt, version=Merchant
	←	callback: PrintReceipt, version=Cardholder
	←	callback: Display("Refund / Approved")
	←	callback: CardStatus(False)
	←	callback: Display("Refund / Approved")
		Refund response is OK

13.8 Reversal of the last transaction

The last approved transaction can normally be reversed. There is no scope for cardholder interaction here, and the client application will not have to supply any additional information.

Client		Terminal
ReverseLastTransaction	→	
	←	callback: PrintReceipt, version=Merchant
	←	callback: PrintReceipt, version=Cardholder
	←	callback: Display("Reversal / Approved")
	←	Reversal response is OK

13.9 Enable the card reader

The client application can enable the card readers (chip card and magnetic stripe readers only) outside of a transaction. This allows the customer to present a card ahead of a transaction being started, allowing the client application more time to establish what kind of transaction is needed.

This can also be used to read bonus / loyalty card information for administrative purposes.

Client		Terminal
EnableCardReaders with enable = True	→	
	←	EnableCardReaders response OK
	←	callback: Display("New customer / Waiting for card")
	←	callback: CardStatus(True)
	←	callback: Display("Please wait")
	←	callback: CardAccepted("541333******0020")