MEI® BNR

Banknote Recycler

BNR API for dotNET

Programmer's guide

Release G8

Published by:

Crane Payment Innovations
P.O. Box 2650
Ch-1211 Geneva 2
Switzerland

MEI BNR - Banknote Recycler - BNR API for dotNET Programmer's guide

Part Number: This edition	**67277 7 044** July 2018	**Release** G8
ECN number	5 000 000 20702	
File reference	BNR API for dotNET 67277-7-044.docx	

VERSIONS

Release G1

First official release.

Release G2

Updated chapters:

- "Managing USB exceptions".
- "How to know if change is available?"

List of USB pipes used by the BNR.

Release G3

Updated chapter “How to select which banknotes to dispense ?”.

Release G4

Description of asynchronous and synchronous methods.

Updated chapter “BNR method summary” with methods Bnr::CancelWaitingCashTaken, Bnr::Eject, Bnr::Reject, Bnr::Retract.

Updated chapter “Intermediate events definition” with intermediate event code BccInserted.

BNR Statuses updated with Barcode reader module.

Release G5

Description of the properties, methods and event for the preventative maintenance.

Updated file structure for the API in chapter 1.2 The BNR API for dotNET.

New file BnrCtlDotNet40.dll added to chapter 2.1 BNR SDK Concepts.

Updated chapter 2.3 Module methods summary with new or missing methods.

Updated chapter 2.2 BNR method summary with the new method Bnr::SaveHistoryReport.

Updated chapter 2.6.2 Error codes and events by functions.

Release G6

Updated copyright with CPI information.

Fixed issues in “2.6 Error codes” chapter.

Fix typos.

Release G7

Correct chapter 9.12 How to work in a degraded mode as locking a module requires a call to Bnr::ConfigureCashUnit().

Updated chapter 2.2 BNR method summary with the new methods.

Release G8

Updated chapter 2.2 BNR method summary with the new methods related to BNR Advance™.

Updated chapter “BNR Statuses” with MM Advance™ specific elements.

TABLE OF CONTENT

1 Introduction

The BNR API for dotNET is Software Development Kit (SDK) for OEM's who integrate MEI BNR into an automated payment machine based on Windows platform. It is a set of libraries to help Integration Software Engineers to drive the MEI BNR.

1.1 About this document

This document is intended for Project Managers and Integration Software Engineers of OEM's (Original Equipment Manufacturer) integrating a MEI «Banknote Recycler» («BNR») into an automated payment machine.

This document contains:

- a description of the implementation of the «Banknote Recycler»
- the principles for the use of the «Banknote Recycler» API
- the necessary information for the management
- reference data of the «Banknote Recycler» API

1.1.1 Conventions used in this document

Typeface Style	Used for:
`Monospace font`	C code and literal values, such as function names.
`Monospace bold`	Points of interests in code samples.
`Monospace italic`	Variables and placeholders.

1.1.2 Contact

Any query related to the release of this product should be addressed to:

Crane Payment Innovations
Technical Support
P.O. Box 2650
Ch-1211 Geneva 2
Switzerland

Tel.: +41 22 884 0505
Fax.: +41 22 884 0504

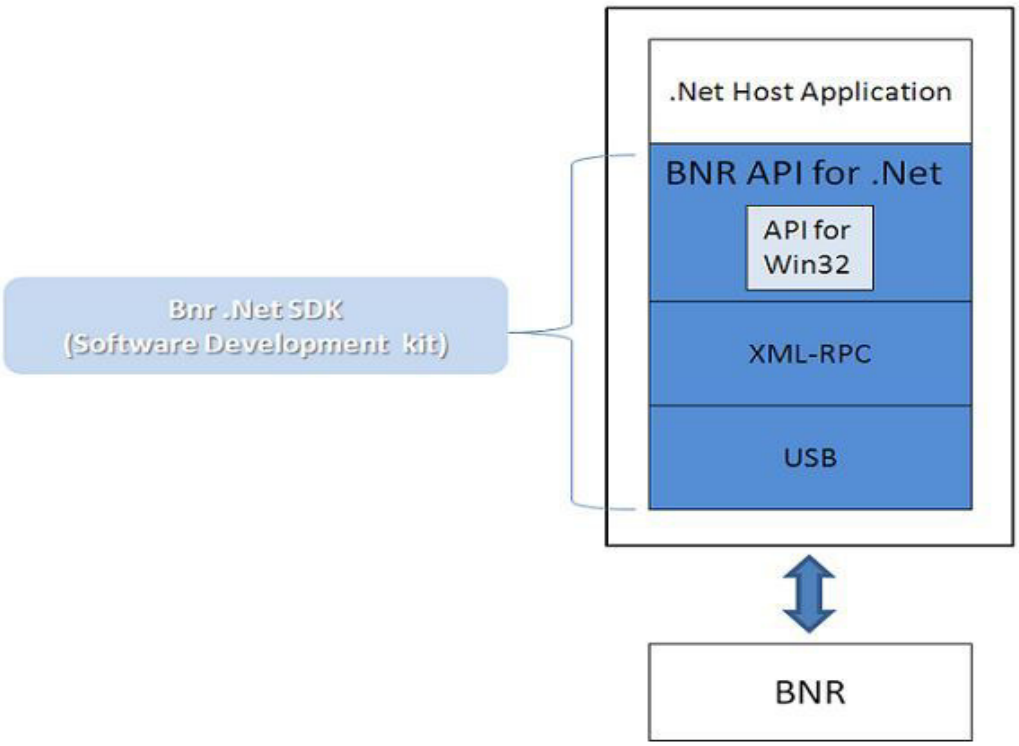
1.2 The BNR API for dotNET

The SDK contains these elements (paths are relative to the install location):

BNR_SDK\Documentation\	All the documentation for the SDK.
BNR API For dotNET 67277 7 044 G*x*.pdf	This programming guide.
bnrCtldotNET-doc.zip	The complete API reference in HTML format.
licenses.txt	Information about open source licenses.
MPL-1.1.txt	Mozilla Public License
BNR_SDK\CommonFiles\x86	Executable and libraries for 32 bit operating systems
BnrClientDotNet.exe	Client application to demonstrate the API.
BnrClientDotNet40.exe	Client application to demonstrate the API.
BnrCtlW32.dll	BNR API library.
BnrXmlRpc.dll	Xml-Rpc parser.
XmlParse.dll	Xml parser.
BnrCtlDotNet.dll	BNR dotNet API
BnrCtlDotNet40.dll	BNR dotNet API
XmlTok.dll	Xml library
BNR_SDK\CommonFiles\x64	Executable and libraries for 64 bit operating systems. File list is the same as …\x86
BNR_SDK_HOME\BNR Usb Driver	USB kernel driver for BNR.
BNR_SDK\vcredist\	Installer for minimum C runtime version required to use the SDK.

2 Using the BNR API

2.1 BNR SDK Concepts



These are the standard application layers defined by MEI. The host application 2 will communicate with the **BNR API for dotNet** through the **BNR API for Win32**. Documentation about the BNR API for Win32 is available.

The DLL BnrCtlDotNet uses Managed C++ and allows calling functions with dotNET technologies.

The BNR API for dotNET needs 6 files to work properly:

Files	Description
BnrXmlRpc.dll XmlParse.dll XmlTok.dll	DLLs containing the xml-rpc functions
BnrCtlW32.dll	The BNR API for WIN32 contains the functions to manage the communication between the host application and the BNR (only needed for the standard application layers choice).
BnrCtlDotNet.dll	The BNR API for .NET using .NET framework prior to 4.0. This is a wrapper of the BNR WIN32 API.
BnrCtlDotNet40.dll	The BNR API for .NET using .NET framework 4.0 or latter.
BnrCtl.ini	File with the configuration for the logs files

BNR USB driver, provided with BNR SDK must be installed prior to use the API.

2.2 BNR method summary

Command Name	Description
Bnr::AddDenomination	Adds denomination to the BNR.
Bnr::Cancel	Attempts to cancel the current asynchronous operation.
Bnr::CancelWaitingCashTaken	Asks the BNR to stop waiting for cash removal at the Bezel if any.
Bnr::CashIn	This command will cause inserted notes to be moved to the escrow.
Bnr::CashInEnd	This command is used to end the cash in transaction.
Bnr::CashInRollback	This command is used to roll back the cash in transaction.
Bnr::CashInStart	This command is used to start the cash in transaction with the BNR.
Bnr::Close	Ends the communication with the BNR and terminates the thread that has been started by a previous `bnr.Open()` call.
Bnr::ConfigureCashUnit	Configure the BNR Cash Unit available in the system.
Bnr::DeleteDenomination	Delete a denomination in the BNR.
Bnr::Denominate	Asynchronous command to determine if the amount requested by value or by bill list, is available for dispense.
Bnr::Dispense	Dispenses the amount requested by value or by bill list.
Bnr::Eject	Force cash that has been presented to be ejected from the bezel.
Bnr::Empty	Empty a recycler or loader cash unit in the cashbox.
Bnr::GenerateAuditReport	Create a file with the data of history and report command.
Bnr::Open	Issues a request through the USB to determine if a BNR is connected, and if a BNR is connected and powered, starts the thread to monitor the BNR and to communicate with it.
Bnr::OpenWithSerialNr	This is an overloaded member function of `bnr.Open()`, provided for convenience.
Bnr::Park	Prepares the BNR for shipment.
Bnr::Present	This command activates the presentation of the cash.
Bnr::PreventativeMaintenanceCompleted	Indicates to the BNR that the Preventative Maintenance (PM) operation of the specified level was completed.
Bnr::Reboot	Reboots the BNR.

Bnr::Reject	Clear the intermediate stacker(escrow) during a Dispense transaction.
Bnr::Reset	This method is used to put the BNR into a defined operational state.
Bnr:: ResetCashboxCuContent	Resets counters in cashbox. It resets Cashbox PCU+LCUs bill count (+initialCount).
Bnr::Retract	Force cash that has been presented to be retracted.
Bnr::SaveHistoryReport	Saves an HistoryReport inside the BNR.
Bnr::SelfTest	Allows BNR to perform self tests in order to stay operational. Available only when BNR's test mode is set to slave (not recommended).
Bnr::SetCapabilities	Sets the BNR capabilities. Those settings are persistent on Power-down.
Bnr::SetLoaderCuContent	Sets loader cash unit counts. InitialCount is set to same value as count. Extended dispenseCounters are resetted to zero.
Bnr::UpdateCashUnit	Updates the cash unit in the BNR
Bnr::UpdateDenomination	Updates denomination setting for bill recognition.

2.3 Module methods summary

Module methods name	Description
Module::MaintenanceDone	Indicates to the module that maintenance operation was done.
Module::Park	Prepares one module in the BNR for shipment.
Module::RevertToBoot	Erases the main application of the Module.
Module::UpdateFirmware	Updates the Module Firmware.

2.4 Asynchronous and synchronous methods

Each BNR API method operates in one of two synchronization modes: asynchronous or synchronous. These are described in the following sections.

2.4.1 Asynchronous methods

Asynchronous mode is used for operations which requires banknote handling or may take an indeterminate amount of time to complete. Performing an operation in an asynchronous, as opposed to a synchronous, mode allows the application to operate in Windows' native event-driven, message-based manner. The processing of an asynchronous request (e.g. `Bnr::CashIn()`) is as follows:

- The application calls the BNR API method.
- The BNR API calls the BNR.
- The BNR generates a sequence number, the identificationId, assigns it to the request, and returns the identificationId to the BNR API and then to the application, Receiving an identificationId means that the request has been initiated and is being processed.
- On completion, the BNR sends a completion message (OperationComplete) to the BNR API, which posts it to the listener specified by the application using delegate. The message contains an object defining the results of the request, including the identificationId, the status code and the other relevant data.

2.4.2 Synchronous methods and properties

Synchronous mode is used when an operation do not requires a banknote movement to complete. The BNR API does not return the control to the application until the operation has completed, thus synchronous functions are referred to as blocking. Each synchronous call made by an application is translated by the BNR API into an XML-RPC call passed to the BNR.

The processing of a synchronous request (e.g. Bnr::TransactionStatus) is as follows:

- The application calls the BNR API.
- The BNR API translates the request into an XML-RPC call passed to the BNR.
- On completion, the BNR sends the result to the BNR API, which returns the control to the application.

2.5 Events

BNR API for dotNET delivers information to the application by events. **Events** are used by the asynchronous methods and can be sent at any time. The following duties are assigned to the events:

- Notify the application of intermediate results during the running operation (e.g. sending the single keystrokes from a keyboard).
- Notify the application of asynchronous operation completion (i.e. track read). This may be successful completion as well as abortion of the operation due to an error.
- Inform the application of status changes (e.g. busy, offline) and special conditions (e.g. threshold values reached such as paper low)

To satisfy the above duties the following categories of events exist:

1) The *IntermediateEvent* (I) is sent whenever meaningful intermediates result is available for the running asynchronous operation.
2) The *OperationCompleteEvent* (OC) is sent whenever an asynchronous operation is completed. The return code depends on whether the operation was successful, partially successful or with a failure. For the operations that return data it is possible to have several sub-classes of this event (containing the additional data and its access methods). They all start with the letters “OC”, e.g. OCReadTrackEvent.
3) The *StatusEvent* (S) is sent whenever device status changes.

2.5.1 Intermediate events definition

Intermediate Event code	Definition
InputRefused	At least one banknote was not recognized during a cashIn operation and has been returned to the reject slot.
SubCashIn	This event is generated when one of the sub-cash-in operations during a permanent cashIn has finished successfully. The data field contains a cashOrder object.
BccInserted	This event is generated when a coupon with barcode has been inserted and recognized during a cash in transaction, the BNR then waits for the RecognitionResult. The data field contains a string with the barcode number.

2.5.2 Status events definition

Status Event code	Definition
CashAvailable	Cash is available at the BNR exit slot.
CashTaken	Cash has been removed from the BNR exit slot.
CashUnitChanged	cashUnit changed.
CashUnitConfigurationCanged	The cashUnit configuration was changed.
CashUnitThresholdChanged	A cashUnit threshold was changed.
DeviceStatusChanged	Device status changed.
MaintenanceStatusChanged	The Maintenance status has changed for one or more modules.

2.6 Error codes

When a function is called, the BNR API returns a BnrXfsErrorCode. This value contains a number which represent the result of the method call.

If the return is higher than zero (return > 0):

This represents the Id of the asynchronous method called.

If the return is equal to zero (return = 0):

This means that it was a synchronous operation and that it has been completed successfully.

If the return is between -1000 and –1 (-1000 **<** return **<=** -1):

The positive value of this number represents the *API error code.*

If the return is between –10000 and –1000 (-10000 **<** return **<=** -1000):

The positive value of this number represents the *XFS Exception*. Eg : -1015 → 1015 →Illegal

If the return is **less or equals** –10000:

The positive value of this number represents the USB Exception.

2.6.1 XFS error code definitions

Error Code	Definition
Successful	Operation completed without error.
Busy	The BNR is busy (moving or about to move bill).
Cancelled	The operation was cancelled by the application via a `Bnr::Cancel()` call or by a user action (door opening).
Illegal	Depends on the method call.
NoHardware	The targeted module is missing in the BNR and the function cannot be achieved successfully.
NotSupported	Operation not supported by the BNR. This may typically happen while the mmMainBoot is running.
ParameterInvalid	A parameter of the method call is illegal.
InvalidMixNumber	The number refers to an undefined mix table or mix algorithm.
CashInActive	The BNR has already a `Bnr::CashInStart()` command issued.
NoCashInStarted	`Bnr::CashInStart()` was not called.
NoBills	There were no bills on the stacker present.
TooManyBills	The request would require too many bills to be dispensed.
NotDispensable	The amount cannot be dispensed.
CashDeviceError	An unspecified error occurred.
CashUnitError	A selected cashUnit caused an error.

2.6.2 Error codes and events by functions

OperationID	Methods								
6117	Reset								
6121	CashInStart								
6122	CashIn								
6124	CashInRollback								
6123	CashInEnd								
6107	Denominate								
6108	Dispense								
6126	Present								
N/A	Cancel								
Error codes and Events									
	Successful commands								
0	Successful	X	X	X	X	X	X	X	X
	Errors and Exceptions								
1010	Illegal	X	X	X	X	X	X	X	X
1018	ParameterInvalid		X	X					
1021	Cancelled	X	X				X		
1022	NotSupported	X	X	X	X	X	X	X	X
6085	InvalidMixNumber		X	X					
6072	CashinActive	X	X					X	
6089	NoCashInStarted				X	X	X		
6088	NoBills	X	X	X					
6092	TooManyBills		X						
6087	NotDispensable		X	X					
6073	CashDeviceError	X	X		X	X	X	X	
6074	CashUnitError	X		X	X				
	Intermediate Events								
6209	InputRefused						X		
	SubCashin						X		
	Status Events								
6223	CashAvailable	X	X			X	X		
6192	CashTaken	X	X			X	X		
6153	CashUnitChanged	X	X		X	X	X		X
6155	CashUnitThreshold	X	X		X	X	X		X
6162	DeviceStatusChanged	X	X		X	X	X	X	X
3100	MaintenanceStatusChanged	X	X		X	X	X		X

All method calls may receive any of the API or USB error messages. These messages are described in the API technical reference. For specific method error codes, check the dotNET API technical reference.

2.7 Managing USB exceptions

USB exceptions correspond to error at USB communication level. Often it is the result of spurious EMI perturbations that may result in communication breakdown. In this case, the API starts automatically recoveries. A USB error will be returned to the caller only if all attempts to recover fail.

There are two cases where a communication breakdown can happen:

1. The BNR never receives the call and thus the API does not get the response too.
2. The BNR receives the call, process it and send the response, but the API never receives it.

For all commands, the API waits for a response from the BNR with a timeout of 10 seconds (default value, may be changed in BnrCtl.ini file, key `RECEIVE_METHOD_RESPONSE_IN_SEC` in section `TIMEOUT`).

- If the response is received before the timeout:

  The response is returned to the caller.

- else:

  The API attempts to recover communication (see **2.7.1 Internal recovery procedure**).

  - If the recovery fails:

    The USB error is returned to the caller. All software actions have been tried and a human intervention is required.

  - else:

    The command is processed as usual (see **2.7.2 Synchronous command after successful recovery** or **2.7.3 Asynchronous command after successful recovery**) and the response is returned to the caller.

2.7.1 Internal recovery procedure

To recover the communication with the BNR, the API executes the following sequence (given for information only):

- Call `Bnr::ResetUsbDevice()`
- Call `Bnr::Close()`
- Call `Bnr::Open()`, which cycles USB port (simulated a USB connection cycle)
- If `Bnr::Open()` is successful:

  End of the procedure with success.

- else:

  Call `Bnr::KillUsbAndReload()` for this BNR only.

  - If `Bnr::KillUsbAndReload()` is successful:

Call `Bnr::Open()` and end of the procedure.

- else:

  End of the procedure with USB error.

The call to `Bnr::KillAndReload()` can be disabled using the BnrCtl.ini file, key `ALLOW_KILL_AND_RELOAD = 0` in section `CAPABILITIES` (see BNR API for Win32-Linux Technical Reference).

2.7.2 Synchronous command after successful recovery

After the communication is successfully recovered, the API will send again the synchronous command with its parameter and the result is returned to the caller.

Special cases:

Due to the behavior of the BNR, 2 synchronous commands must be handled differently:

- **Bnr::Reboot** – If the BNR is processing the first call while the API is doing the second one, a USB error will be returned. Recommendation: wait for 30 seconds before calling `bnr_Open`.
- **Bnr::UpdateCashUnit** – If the BNR received the first call, a BNR internal bill transfer might already been started to refloat the recycler, while the API is doing the second call and an `XFS_E_BUSY` will be returned. Recommendation: wait for the end of the transfer by polling the BNR every second with `Bnr::TransactionStatus`.

2.7.3 Asynchronous command after successful recovery

After the communication is successfully recovered, the API will do several calls to determine if the BNR received the first call or not.

If the BNR did not receive the first call, the asynchronous command is repeated and the result is returned to the caller as usual.

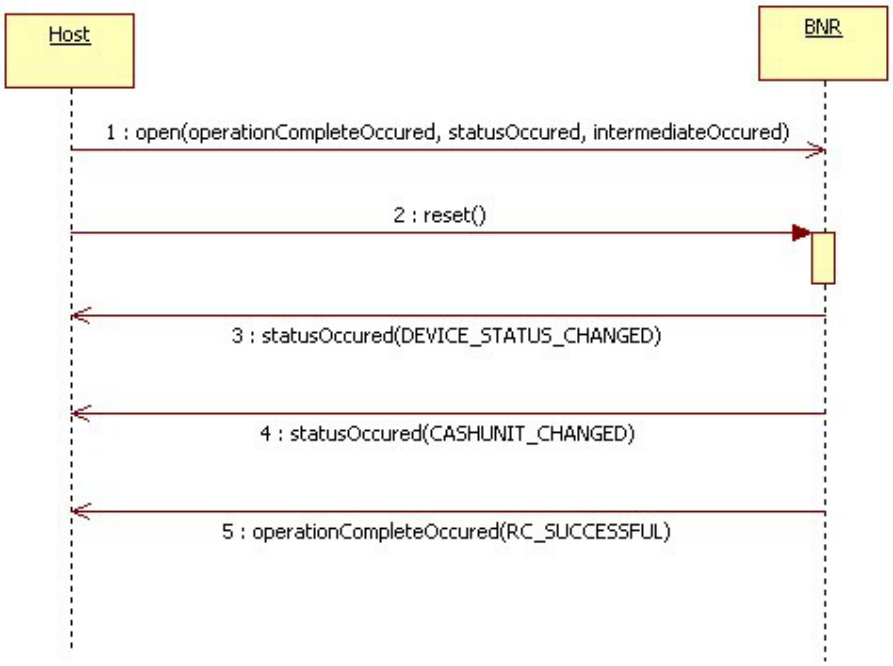
If the BNR is already processing the first call, the API will return the `IdentificationId` to the caller as usual. In this case, the call is successful and the caller must wait for the command completion, notified with the OperationComplete event.

Warning :

With BNR firmware prior to 1.4.0, if the BNR response has been lost during the communication breakdown, it will return 0 as `IdentificationId` value.

3 BNR and communication initialization

This chapter describes the basics steps to establish communication with the BNR and put it in a determinate state.



The OperationCompleteOccured contains an extendedResult filed that informs about the final result of the command. If the result is Success, and the extendedResult error code is equal to zero, the BNR is ready to accept commands from the host.

4 Cash acceptation

This chapter describes how to make the BNR accept cash from customers. For the cash acceptation, we assume that an `Bnr::Open()` and a `Bnr::Reset()` commands have been issued and completed successfully.

The cashIn command is used to make the BNR accept banknotes. It has to be use only after a successful call to `Bnr::CashInStart()`. The call to cashIn causes the BNR inlet LEDs to turn green.

The BNR can accept cash in three different ways.

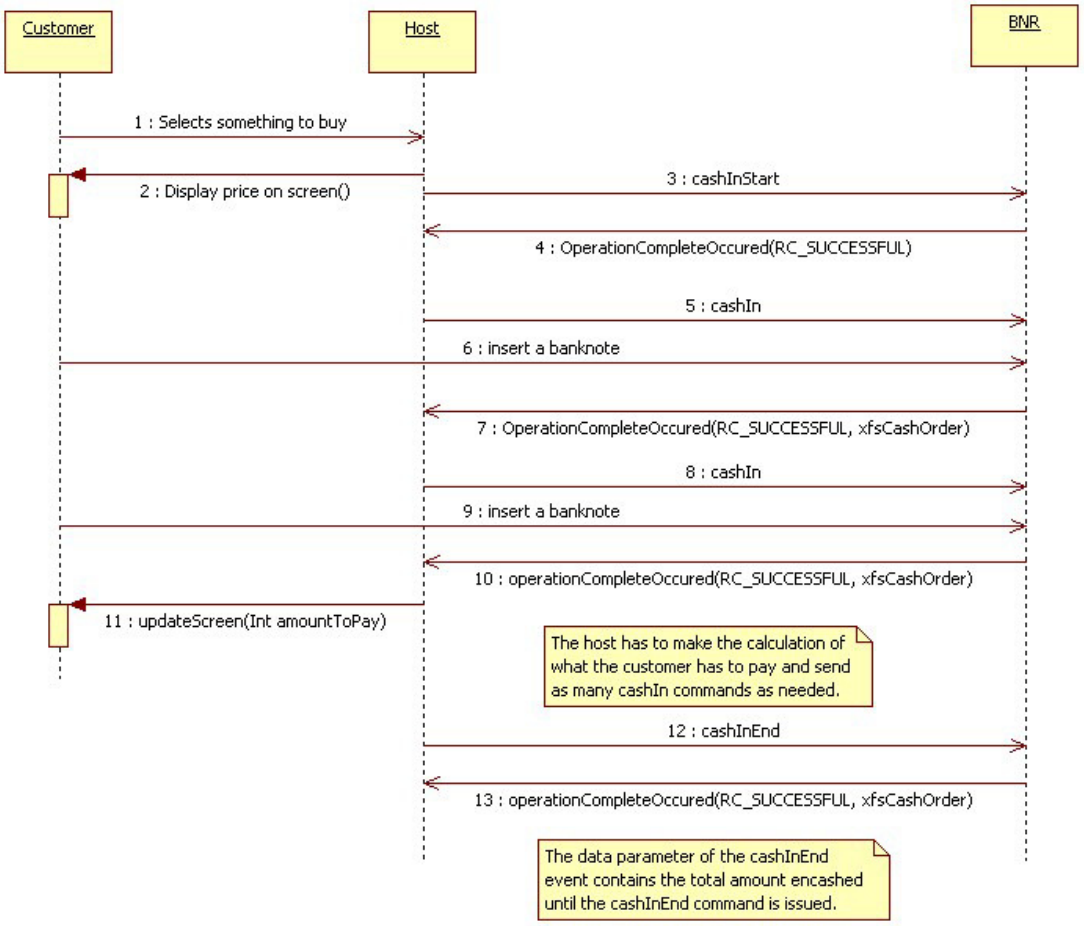
- **`Bnr::CashIn`** method call issued **without parameters**. In this case, the BNR will accept one banknote and inform about the document inserted with an OperationCompleteOccured event. To accept more than one banknote, it is necessary to make a loop and calculate the total amount accepted.

- **`Bnr::CashIn`** method call issued **with amount parameter**. In this case, the BNR will accept cash until the amount introduced is equal or higher to the amount parameter send through the cashIn command. The BNR will inform through an IntermediateEvent the value of the last banknote accepted. Once the amount inserted is equal or higher to the amount parameter, the BNR will fire an OperationCompleteOccured with the total accepted.

- **`Bnr::CashIn`** **with parameters** set to **null**. In this case the BNR will go to a permanent cashIn mode. It will inform of the value of each banknote accepted with an IntermediateEvent. Once the host decides that the amount accepted by the BNR is sufficient, it has to issue a `Bnr::Cancel()` command. The BNR will then respond with an OperationCompleteOccured event containing the total accepted.

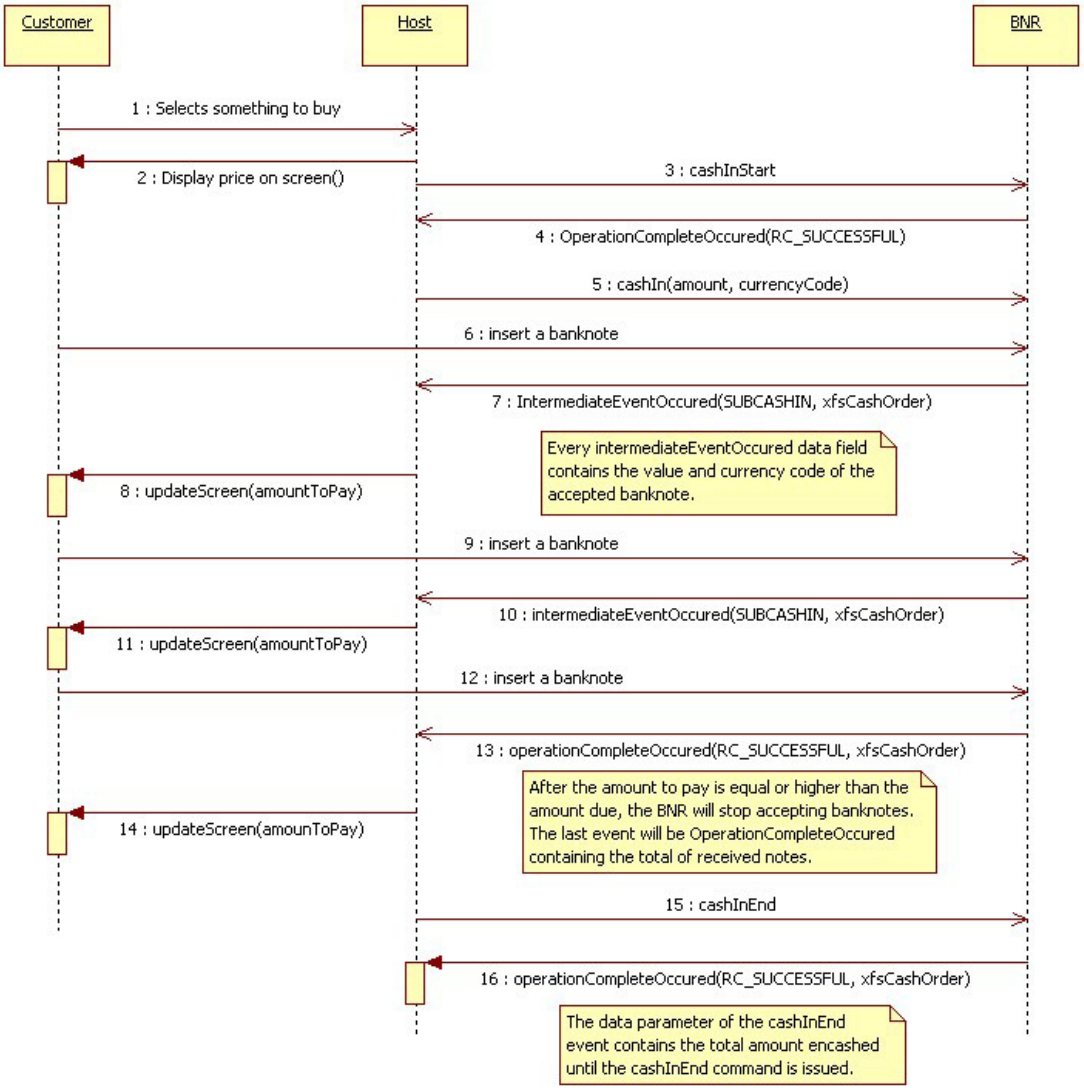
N.B : This diagrams assumes that the “Host” already completed the initialization sequence. For more details, refer to chapter .4.

4.1 cashIn without parameters :



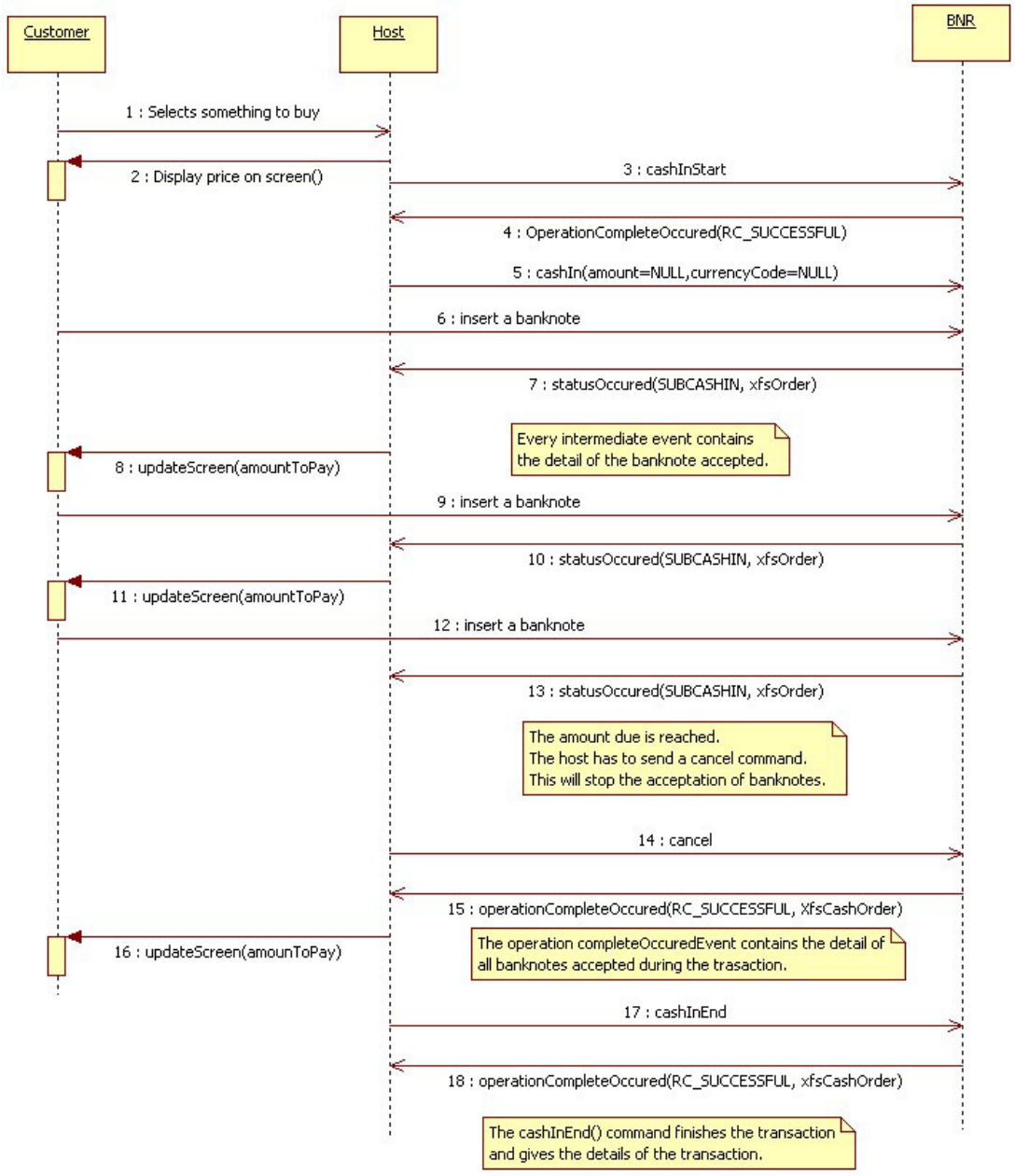
This example shows how to send a simple `Bnr::CashIn()` method call **without parameters**.

4.2 cashIn with amount and currency code parameter



For this method call, the host sets the amount parameter.

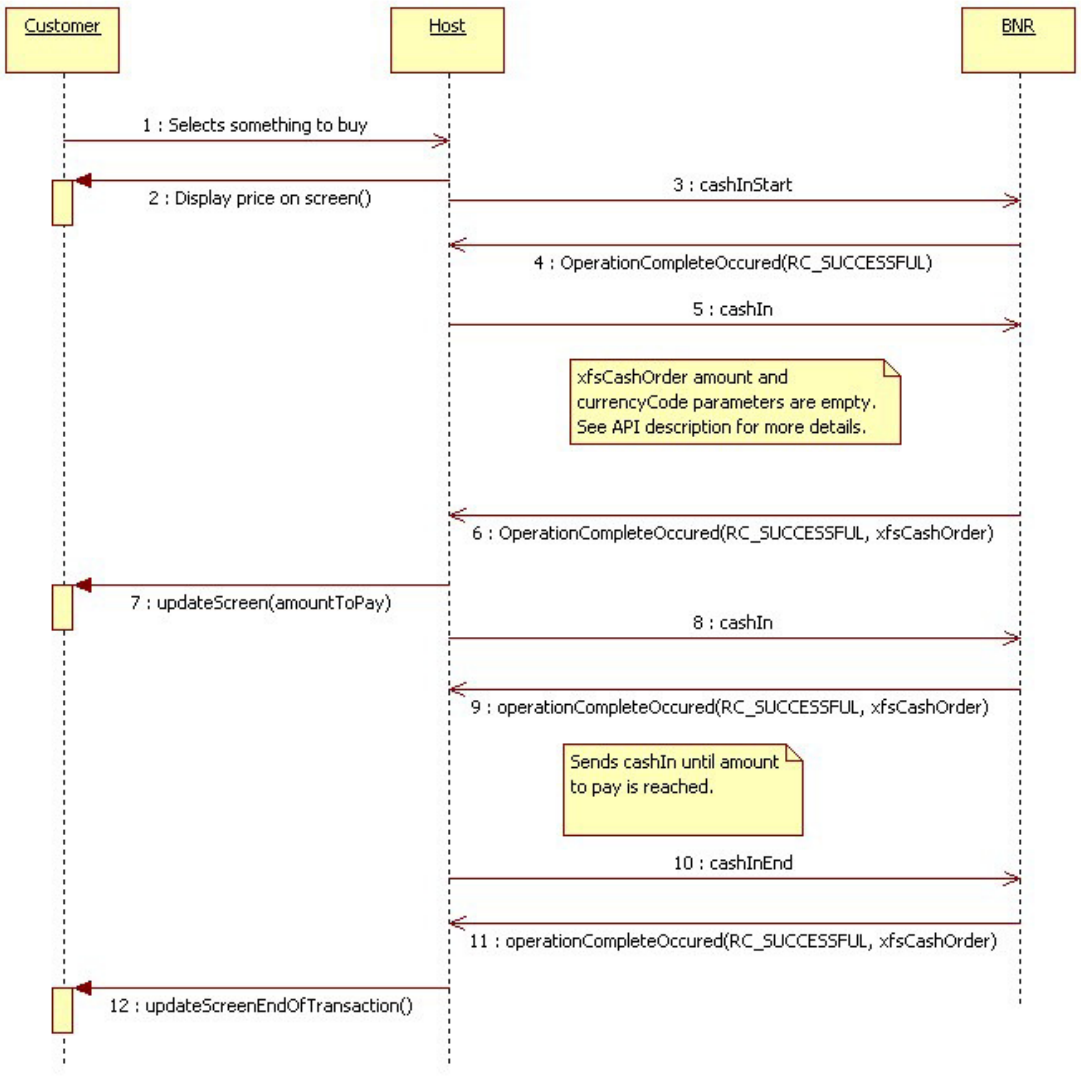
4.3 cashIn with parameters set to null



5 Finishing a transaction

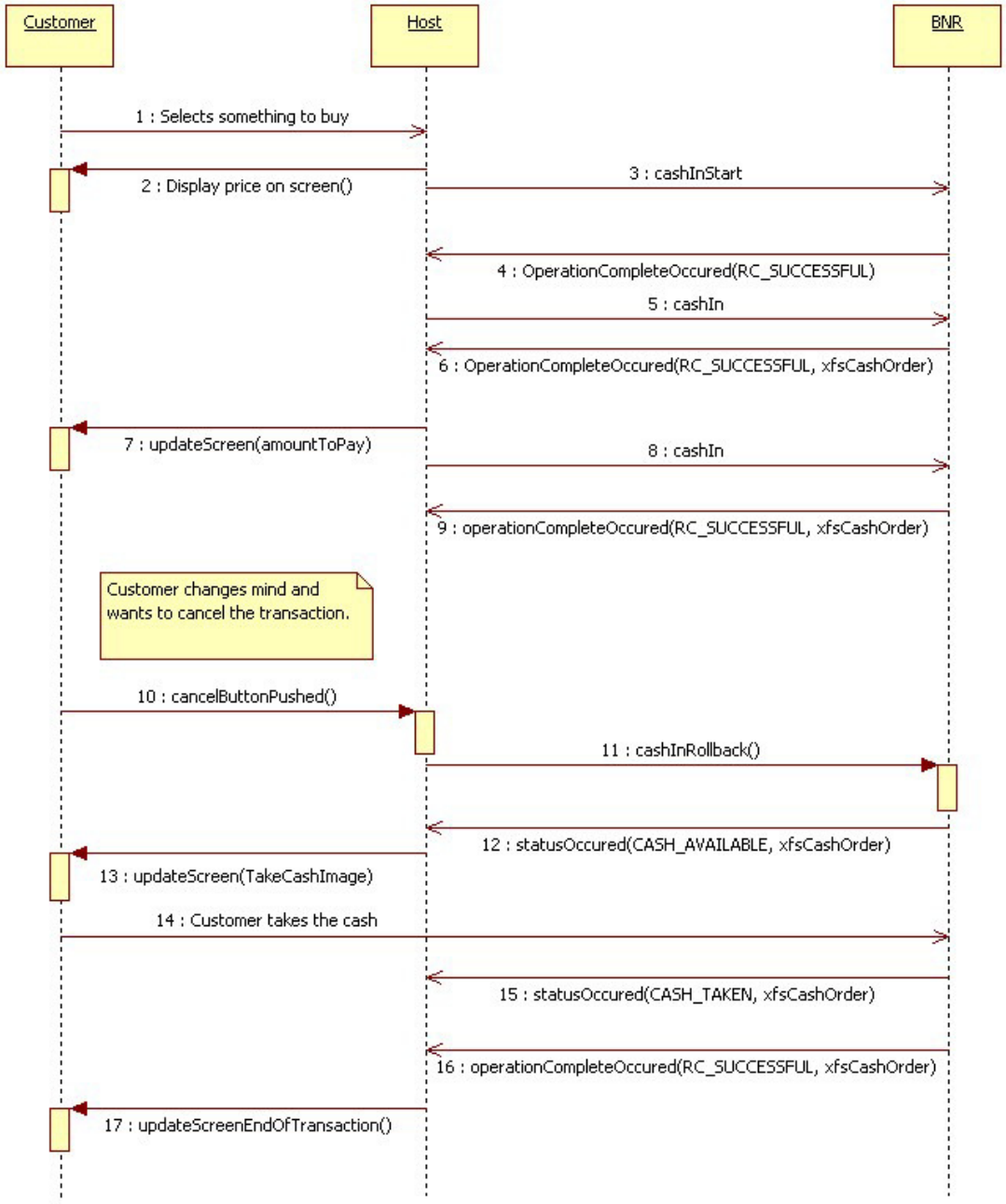
The BNR has escrow capabilities that allow the host to either rollback the customer bills or to finish the transaction. The escrow has a capability of 15 banknotes. If the customers cancels the transaction and as long as the host didn't send a `Bnr::CashInEnd()` command, the BNR can rollback the exact same bills that have been introduced by the customer. In case of sending `Bnr::CashInEnd()` command, banknotes stored in the recyclers in escrow position will remain in the recyclers but counted as deposit. Those from the mainModule escrow will go to the cashbox.

5.1 Transaction without rollback.



In this example, the customer puts the exact amount he has to pay. By the end of the transaction, the Host has to send the `Bnr::CashInEnd()` command to secure the cash in the BNR (after `Bnr::CashInEnd()` command, no `Bnr::CashInRollback()` command is allowed).

5.2 Transaction with rollback

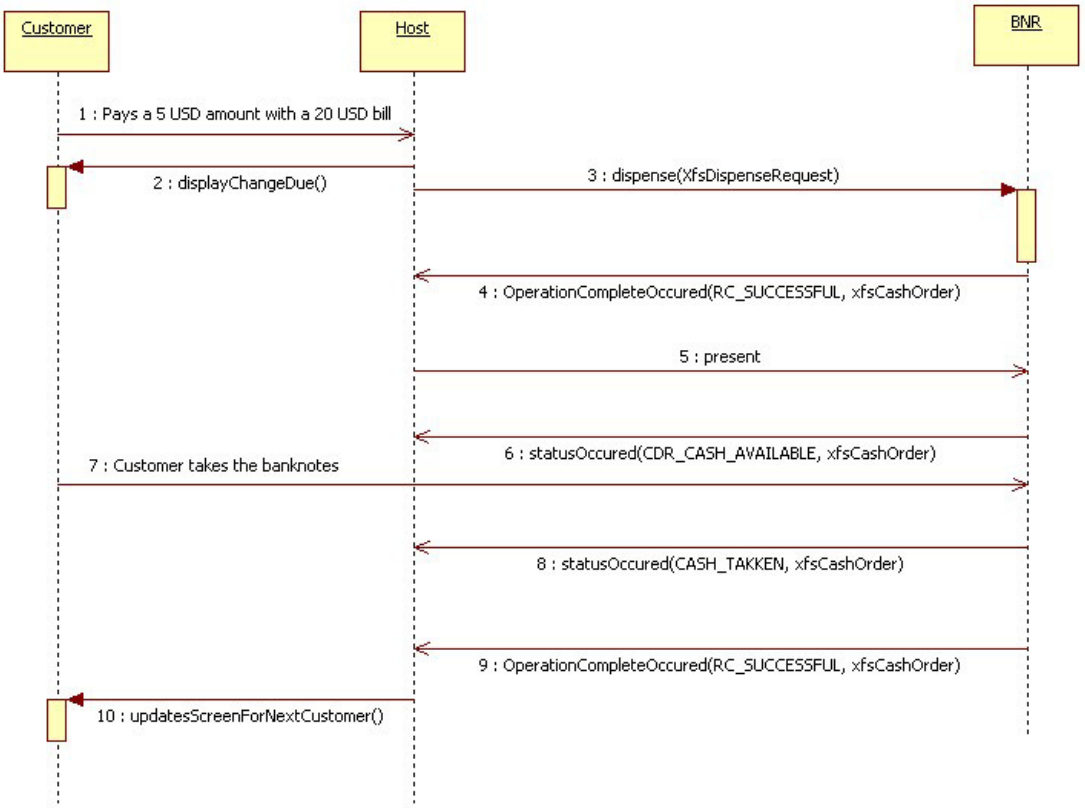


In this case, the customer wants to cancel the transaction. By sending a `Bnr::CashInRollback()` command, the BNR will roll back all the banknotes accepted since the last `Bnr::CashInStart()` command was issued. The customer will get the exact same banknotes as he puts in the BNR.

6 Dispensing cash

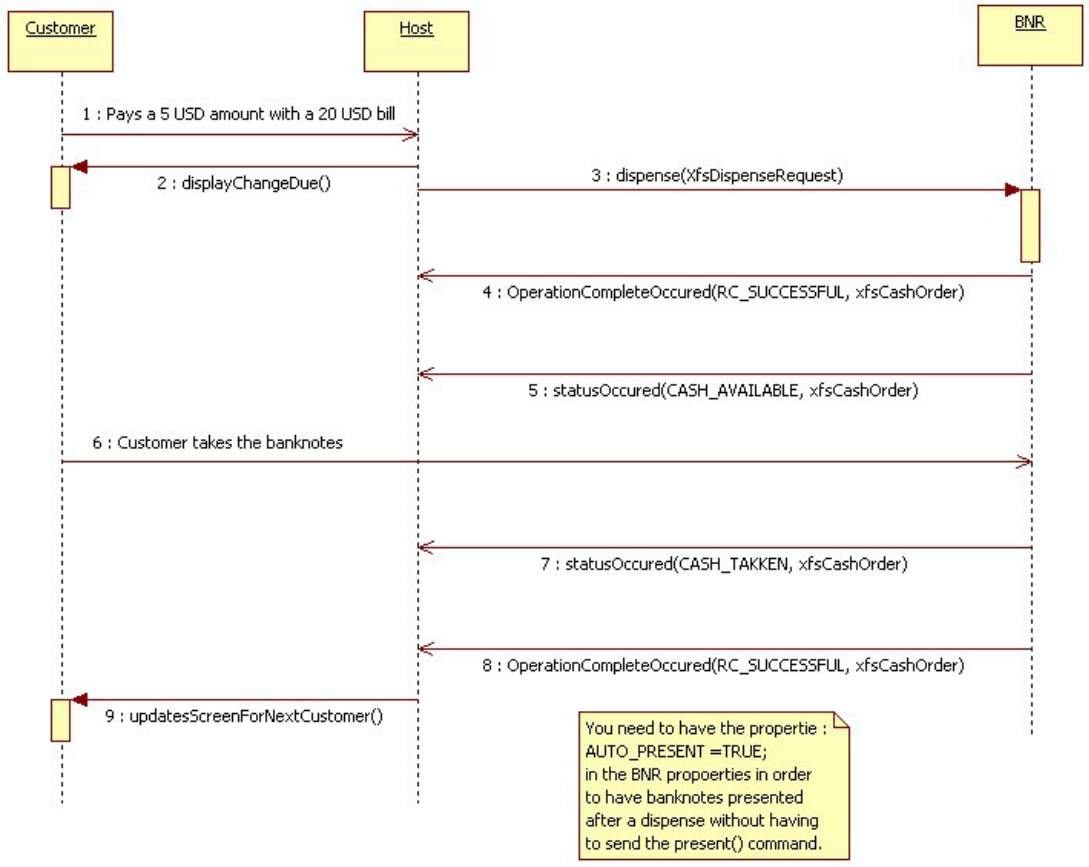
The BNR can dispense cash using the banknotes accepted during previous transactions. It can dispense up to 15 notes at the time. Depending on the BNR configuration, the host may need to ask the BNR to present the cash to the customer.

6.1 Dispense and Present



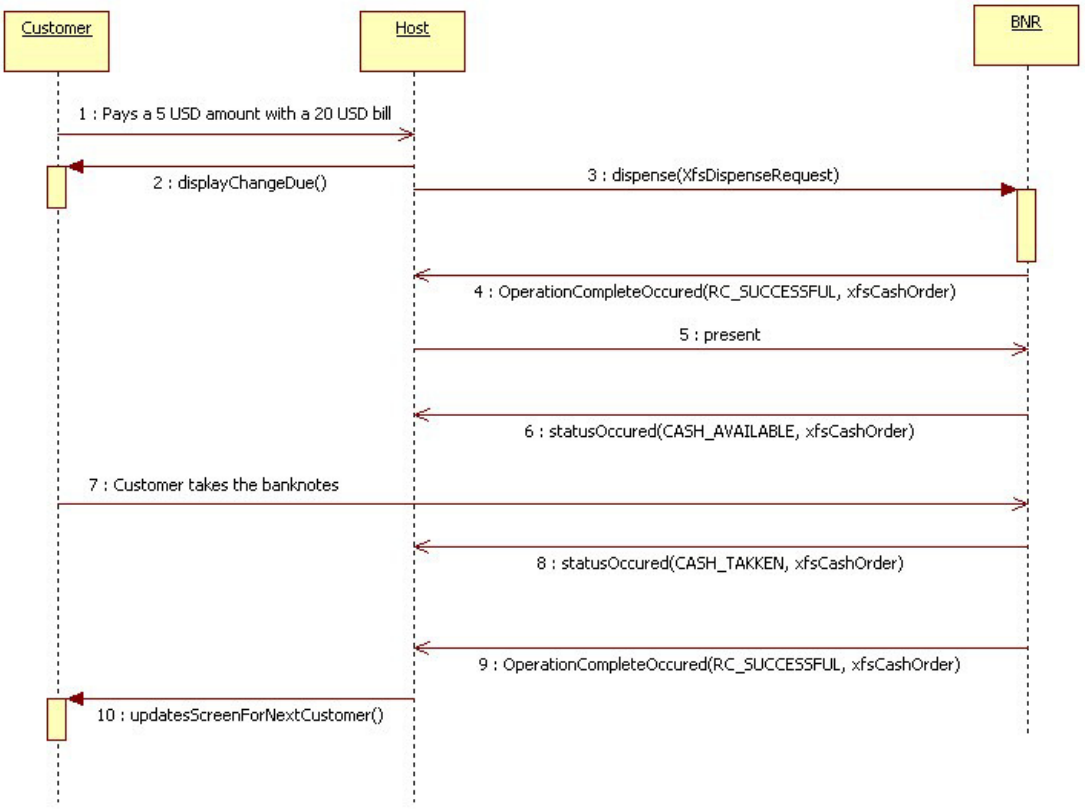
The BNR will move the banknotes to the bundler in the mainModule. It will then wait for the `Bnr::Present()` command to be sent by to host to present the cash to the customer.

6.2 Dispense with autoPresent :



If the autoPresent option is set, the host doesn't need to send the `Bnr::Present()` command to the BNR in order to have the cash presented to the customer. To activate the autoPresent option, see `Bnr::Capabilities` property function description in the API description document.

6.3 Dispense of a single banknote



When dispensing only one banknote, the BNR will only move the banknote after the `Bnr::Present()` command call. If `autoPresent` is set, the BNR will move immediately the banknote.

7 Getting started: A tutorial example

This chapter will help you getting started with the BNR behavior by walking through the creation of a simple client application. This application will be able to make the BNR operational, accept note in the BNR for some amount and give change if needed.

All examples of this document will use the Microsoft Visual Studio 2005 tool. However, it can be easily adapted to other programming environment.

7.1 Creating a BNR client application

A BNR client application requires the C headers files and the libraries provided by the BNR SDK. For the purpose of this tutorial, we will create a client application working in console mode.

7.1.1 Create the Visual Studio project

1. Launch MS Visual Studio and choose “File\New\Projects…”
2. In “Visual C#” select “Windows” as project type and "Console Application” as template.
3. Using the “Browse” button, select the parent folder of your project. As project name, use “BnrClientApplication”
4. Uncheck the option “Create directory for solution” and click “OK” button to create the project.
5. Using F7 key, build the project in debug mode for the first time. You should have no error.

7.1.2 Add BNR SDK files

1. Copy from the BNR SDK, the `XmlParse.dll`, `XmlTok.dll`, `BnrCmlRpc.dll`, `BnrCtlW32.dll` and `BnrCtlDotNet.dll` and place them into the `BnrClientApplication\bin\debug` folder, along with the `BnrClientApplication.exe` file.
2. Add also the library dependency in the BnrClientApplication project properties.
3. Edit the `BnrClientApplication.cs` and add the following code:

```
#using Mei;

namespace BNRConsoleApplicationDotNET{
      class Program{
            static void Main(string[] args){
                  Bnr bnr = new Bnr();
            }
      }
}
```

4. Build the solution using F7 key. You should have no error.

7.2 Connecting to the BNR

Before working with the BNR, it is required to open the communication with it. There is 2 ways to do it: by selecting the first connected BNR on the host machine with `bnr.Open()` or by selecting a BNR serial number in a list with `bnr.BnrSerialNrList;` and `bnr.OpenWithSerialNr(String^ bnrSerialNr).`

Use this step to open a communication with the first BNR:

1. Edit the BnrClientApplication.cpp file and in the `Main()` function, call the `bnr.Open()` function:

```
bnr.Open();
```

2. After having opened a communication with a BNR, no other application can use it. So when do you finish to work with the BNR, do not forget to close the communication;

```
bnr.Close();
```

7.3 Making the BNR operational

This example shows how to request to the BNR to start a reset cycle in order to be back to operation. This is needed just after the BNR power up, but also after BNR opening, etc… In other words, every time the status from `bnr.Status` is different than `OnLine`.

1. Edit the BnrClientApplication.cs file and in the `Main()` function, change the previous lines in order to close the API only if the call to `bnr.Open()` was successful:

```
static void Main(string[] args){
      try{
           bnr.Open();
           bnr.Close();
      }
      catch(Mei.Bnr.BnrUsbException e){
           System.Console.WriteLine("Prob conexion USB ...");
           System.Console.WriteLine(e.ToString());
      }
}
```

2. Before the `bnr.Close()` functions, inserts a call to a new function called `makeBnrOperational()`:

```
static void Main(string[] args){
      try{
           bnr.Open();
           makeBnrOperational();
           bnr.Close();

      }
      catch(Mei.Bnr.BnrUsbException e){
           System.Console.WriteLine("Prob conexion USB ...");
           System.Console.WriteLine(e.ToString());
      }
}
```

3. Create a function named `makeBnrOperational()`:


```
static void makeBnrOperational(){
          //body of the function
}//makeBnrOperational
```

4. In the function's body, use the `bnr.Status` to determine if the BNR is operational and if needed, request for a `bnr.Reset()`. Place those lines of code in the body of your function:

```
//body of the function

Mei.Bnr.Status bnrStatus = bnr.Status;
if (bnrStatus.DeviceStatus != DeviceStatus.OnLine){
  bnr.Reset();
}//if
```

5. Save your changes and compile your application.
6. Run the application. If a BNR is connected and under power, the exit code should be 0 and the BNR should start a reset cycle.

If you run the application a second time right after the first one, you will notice that the BNR did not start a second reset cycle. This is because it is already operational.

7.4 Receiving events

From the previous steps, the application exits before the end of the reset cycle. This is because `bnr.Reset()` is an asynchronous command. The end of the command is determined by the OperationComplete event, sent by the BNR.

Now, we are going to add code to receive the BNR events and to be able to wait for them.

1. Edit the BnrClientApplication.cpp file and define `operationCompleteEvent` as a global variable:

```
private static AutoResetEvent ev = new AutoResetEvent(false);
```

2. In the `Main()` function create the `OperationCompleteEvent`:

```
static void Main(string[] args){
          Bnr.OperationCompletedEvent += new
          OperationCompleted(Bnr_OperationCompletedEvent);

     try {
          bnr.Open();
          makeBnrOperational();
          bnr.Close();
     } catch(Mei.Bnr.BnrUsbException e){
          System.Console.WriteLine("Prob conexion USB ...");
          System.Console.WriteLine(e.ToString());
     }//try
}//Main
```

3. Create a static functions named `Bnr_OperationCompletedEvent()`, in the file:

```
static void Bnr_OperationCompletedEvent(int identificationId, OperationId
operationId, BnrXfsErrorCode result, BnrXfsErrorCode extendedResult, object
Data)
{
     //body of the function
}
```

4. In the body of the function to synchronize the application with the event, add the following:

```
//body of the function
ev.Set();
```

5. In the `MakeBnrOperational()` function, reset the Windows event before the call to `bnr.Reset()` and then wait for the BNR reset completion event after the call:

```
ev.Reset();
bnr_Reset();
ev.WaitOne(BnrResetTimeOutDelayInMS);
```

7. Define the constant `BnrResetTimeOutDelayInMS` to 60 sec.:

```
static int BnrResetTimeOutDelayInMS = 60000;
```

8. Save your changes and compile your application.
9. Power on the BNR and connect it to the USB port.
10. Run the application. The BNR will start a reset cycle and at the end of the cycle, the application will exit.

A production application must manage the `bnr.Reset()` return code and the risk that the `Bnr_OperationCompletedEvent()` function may never be called within the allocated time (60s in this example) if the USB communication is broken. For more information, see chapter “Managing BNR communication exceptions”.

7.5 Accepting banknotes

The BNR is now operational. These steps describe how to accept banknotes with the BNR.

1. Edit the BnrClientApplication.cs file and define `ocResult` and `cashOrder` as global variables:

```
static BnrXfsErrorCode OCresult;
static XfsCashOrder cashOrder;
```

2. In the function `Bnr_OperationCompletedEvent()`, memorise the `result` parameter and the `cashOrder` for cashIn operation with this code:

```
//body of the function
OCresult = result;
switch (operationId)
{
     case OperationId.Reset:
     if (result == BnrXfsErrorCode.Success){
     cashOrder = (XfsCashOrder)Data;
```

```
        }//if
    }//switch
    ev.Set();
```

3. Define a new function called `AcceptAmount()` :

```
static void AcceptAmount(int amount)
{
      //body of the function
}//AcceptAmount
```

4. In the function's body, reset the Windows event and then start a cashIn transaction with a call to `bnr.CashInStart()` and wait for the operation completion event:

```
//body of the function
int insertedAmount = 0;
bool EventTimedOut = false;
ev.Reset();
OCresult = BnrXfsErrorCode.NotUsed;
bnr.CashInStart();
ev.WaitOne(BnrDefaultOperationTimeOut);

//Accepting banknotes
```

5. To accept banknotes, add code to call `bnr.CashIn()` function until the `amount` is reached:

```
//Accepting banknotes
if (OCresult == BnrXfsErrorCode.Success)
   {
       while ((amount > insertedAmount) && !EventTimedOut)
          {
             System.Console.WriteLine("Please insert the bill(s).");
             ev.Reset();
             OCresult = BnrXfsErrorCode.NotUsed;
             bnr.CashIn();
             if (!ev.WaitOne(BnrDefaultOperationTimeOut) || OCresult !=
BnrXfsErrorCode.Success)
                {
                 EventTimedOut = true;
                 bnr.Cancel();
                 ev.WaitOne(BnrDefaultOperationTimeOut);
                 System.Console.WriteLine("Error during cashIn operation.");
                }
             else
             {
               insertedAmount += cashOrder.Denomination.Amount
             }//if
          }//while

//Ending cashIn transaction

}//if
```

6. When the `amount` is reached, end the cashIn transaction with `bnr.CashInEnd()` call and wait for operation completion:

```
//Ending cashIn transaction
```

```
if (OCresult == BnrXfsErrorCode.Success)
{
    ev.Reset();
    OCresult = BnrXfsErrorCode.NotUsed;
    bnr.CashInEnd();
    ev.WaitOne(BnrDefaultOperationTimeOut);
}//if
```

7. Define the two constants `BnrDefaultCashInOperationTimeOutInMS` and `BnrDefaultOperationTimeOut`:

```
static int BnrDefaultCashInOperationTimeOutInMS = 60000;
static int BnrDefaultOperationTimeOutInMS       = 10000;
```

8. In the class, define the variables `instertedAmount` to know how much cash has been accepted and `amountDue` :

```
static int insertedAmount = 0;
static int amountDue = 1000;
```

9. In the `Main()` function add a call to `acceptAmount()`:

```
//7.3 Making the BNR operational
makeBnrOperational();

//7.5 Accepting banknotes, max 1000 MDU
acceptAmount(amountDue);
```

   The max amount of 1000 MDU is an arbitrary value that may be adapted to the currency available ine the BNR for this example.

10. Save your changes and compile your application.
11. Power on the BNR and connect it to the USB port.
12. Set the requested amount as command line parameter.
13. Run the application. The BNR will start a reset cycle, start accepting banknotes until the requested amount. Then the application will exit.

Another way to accept amount is to use `bnr_CashIn()` with parameter. However, doing it as explained in this example give more control to the application, especially in case of error happening during banknote acceptance.

7.6 Dispensing banknotes

Our application is now able to accept banknotes, but often it would be good to be able to give change to the user. These steps will describe how to do it with the BNR.

1. Edit the BnrClientApplication.cs file and add a new function called `DispenseAmount()`:

```
static void DispenseAmount(int amount) {

  //body of the function

}//DispenseAmount
```

2. In the function's body, reset the Windows event and then start a dispense transaction with a call to `bnr.Dispense()` and wait for the operation completion event:

```
//body of the function

String currencyCode = "EUR";
ev.Reset();
OCresult = BnrXfsErrorCode.NotUsed;
bnr.Dispense(amount, currencyCode);
if (!ev.WaitOne(BnrDefaultOperationTimeOutInMS) || OCresult !=
BnrXfsErrorCode.Success)
   {
      System.Console.WriteLine("Error during Dispense.");
   }//if
else
   {
      //Presenting banknote
   }//else
  ;
}//if
```

3. If the result of the operation is successful, the bundle can be presented to the user with a `bnr.Present()` call:

```
//Presenting banknotes

if (OCresult == BnrXfsErrorCode.Success)
    {
     ev.Reset();
     OCresult = BnrXfsErrorCode.NotUsed;
     bnr.CashInEnd();
     ev.WaitOne(BnrDefaultOperationTimeOutInMS);
     System.Console.WriteLine("amount inserted :  " +
     cashOrder.Denomination.Amount);
    }//if
```

4. In the `Main()` function, define `overdue` variable to calculate the amount of change and add a call to `DispenseAmount()`:

```
static int overdue = 0;

//7.5     Accepting banknotes, max 1000 MDU
AcceptAmount(1000);

//7.6 Dispensing banknotes
overdue = insertedAmount - amountDue;
if (overdue > 0) {
    dispenseAmount(overdue);
}//if
```

5. Save your changes and compile your application.
6. Power on the BNR and connect it to the USB port.
7. Set the requested amount as command line parameter.
8. Run the application. The BNR will start a reset cycle, start accepting banknotes until the requested amount (1000 MDU) and give change if needed. Then the application will exit.

8 Advanced programming

8.1 Understanding the refloat process

The refloat process has been designed to be as transparent as it could be for the host. If needed the process starts automatically after a reset or a present command. It will move banknotes from Loader to Recycler until one of the following conditions is reached, whichever is the first:

- A `Bnr::CashInStart()`, `Bnr::Dispense()`, `Bnr::Empty()` or B`nr::Reset()` command is received from the Host.
- The float level (low level threshold) in the Recycler is reached.
- The Loader becomes empty.
- The BNR state becomes different than operational (`OnLine`).

When this process starts or during its execution, no event is sent to the host. Only when the refloat process stops, then the BNR sends a status event `CashUnitChanged` with data on LCUs and PCUs that have changed. If no cash unit has been changed, no status event is sent.

As for all other bill handling commands, during the refloat process, reading the `Bnr::CashUnit` property will raise a `BnrXfsException` exception with code `BnrXfsErrorCode::Busy`.

During banknote movements (including refloat), a status event `CashUnitThreshold` may be sent by the BNR if one or more PCU thresholds are reached. This is typically the case at the end of the refloat process, if the refloat threshold is reached.

9 How to?

This chapter intends to give answers to some of the most common question you may have during the BNR integration. Some of the questions are related to accomplish different actions with the BNR, others are tips to simplify your integration job.

9.1 How to avoid creating some of the structures?

Some functions in the API share the same parameters types. This can be really useful, in particular when the structure that has to be passed as a parameter is a complex structure

List of functions sharing the same parameters:

Functions	Parameter	Remarques
Denominate Dispense	XfsDenominationItem	You can ask the BNR for change availability with the denominate method and then use the same parameter for the dispensing the change.
GetCapabilities SetCapabilities	Capabilities	Get the capabilities with GetCapabilities, modify the structure and use it with SetCapabilites.
GetDefaultBillRouting SetDefaultBillRouting	BillRoutingTable	Get the bill routing table with GetDefaultBillRouting, modify the structure and use it with SetDefaultBillRouting.
QueryDenominations UpdateDenominations	T_DenominationList	Paramters used in the QueryDenominations can be modified and used in the bnr_UpdateDenominations function call.

9.2 How to know if change is available?

A frequent case in transaction can be described like this:

Customer wants to pay a small amount with a big denomination. At this point in the transaction controller has to make a decision: either encash or roll back the cash because it cannot give the change on the amount inserted. Part of the decision has to be taken depending on the change available in the BNR.

The host can use the `Bnr::Denominate()` function to get information about the cash availability. The good point with `Bnr::Denominate()` is that the parameters are the same as for a `Bnr::Dispense()` function. The denominate function will return the list of banknotes that are available for the specified amount or send an error message. If the response is `Successful`, the host can use the parameter sent in the `Bnr::Denominate()` function to make the dispense.

If not enough change is available in the BNR, the field cashbox indicates the amount that is not available in the BNR. This amount should be delivered to the customer by another mean.

9.3 How to select which banknotes to dispense?

The BNR allows the host to decide how it wants to dispense. Three modes are available. They can be selected using the `algorithm` parameter when calling the `Bnr::Dispense(amount, currencyCode, algorithm)` or by calling `Bnr::Dispense(items)` function.

- Dispense with `MinBills` (if BNR firmware version is lower than 1.4.0, use `MixAlgorithm` instead). This is the easiest way to make a dispense. The host sends an amount to dispense and the BNR will dispense using the best change algorithm. That means that the minimum number of banknotes will be used to reach the amount.
- Dispense with `OptimumChange` (not available with BNR firmware version lower than 1.4.0). The host sends an amount to dispense and the BNR chooses the banknotes to be distributed in order to obtain the total amount in a way that slows down cashbox filling. As long as the low denomination Recyclers are not near to full, change is determined like with the `MinBills` algorithm. But when a Recycler becomes nearly full (5/6 of Full threshold), this algorithm will try to put more of this denomination in the change so that the Recycler doesn't become full and this denomination doesn't start to be cashed.
- Dispense with `Bnr::Dispense(items)`. In this case, the host has to send a list of banknotes (`items` parameter) to the BNR. This may allow the host to dispense three five USD bills instead of one 10 and a one five to make a fifteen USD dispense.

With both dispense parameters, the BNR will send an OperationCompleted event containing the list of banknotes used to make the dispense.

These three modes can be used with the `Bnr.Denominate()` function too.

9.4 How to set denominations in the recyclers?

By default, when downloading a variant into the BNR, the smallest denominations are recycled. In some cases you may need to recycle different denominations. This can be done by using the `Bnr::UpdateCashUnit()` function. For more details, refer to "BNR API for dotNET Documentation".

9.5 How to set a minimum number of banknotes in a recycler?

Setting a minimum number of banknotes in a recycler is the way you can control BNR refloat process. When the BNR reaches the minimum threshold of a specific recycler, it will try to pick banknotes from the Loader (refloat process). To set this minimum number of banknotes in a recycler, you have to use the `Bnr::UpdateCashUnit()` function.

9.6 How to stop the BNR from refloating?

During idle time, the BNR check the minimum threshold of the recyclers and if the recycler which denomination matches with the loader is under the minimum threshold, it begins a refloat cycle that will be stopped only on two conditions.

To stop this cycle, you have to match at least one of these conditions:

- The minimum threshold from the recycler is equal to the count of the recycler.
- The host calls a `Bnr::CashInStart()` method or a `Bnr::Dispense()` method.

In both cases, the BNR will finish the action it was processing and fire an event detailing the banknotes that have been moved. The event will contain the source/destination `UnitId`.

9.7 How to set cashbox counters to zero after emptying?

When removing a cashbox from the BNR, it is mandatory to empty it. Once you remove the banknotes from the cashbox, the counter in the BNR doesn't match anymore with the cashbox content. In order to resynchronize your counters with reality you have to set them to zero. This is done using the `Bnr::ResetCashboxContent()` method. It will set to zero all the counters related to the cashbox, physical and logical.

9.8 How to add bills to the billset list?

To add or update bills to the billset list is important for security and performances of the BNR. Adding or updating banknotes in the BNR can be done using the `Bnr::AddDenomination()` method.

9.9 How to get the list of the BNR modules?

To perform some actions like getting the status of a module or updating a firmware, you may need the list of modules presents in the BNR. This can be accomplished by issuing the `Bnr::Modules` property.

9.10 How to know the BNR or a module status?

In some cases, knowing the status of a module can be useful for the Host. In order to know what is the next action to take. This can be done with by checking each module property. Accessing to the property will inform the Host about the specific module status (operational, functional or error status.).

9.11 How to find which module is in default?

When receiving a status event from the BNR, it may contain information about the new status of the BNR. If the status changes to a HardwareError, it may be interesting for the host to know which module is in default. To know which module is in hardware error, you have to make a loop, and check all the module properties you want to test.

9.12 How to work in a degraded mode?

Depending on the module that is in hardware error, you can keep working with the BNR in a degraded mode. Modules like recyclers or loaders are not mandatory for the BNR functionality. Others like spine or mainModule have to be in a working condition. In order to make the BNR work with a defective module that is not mandatory, you have to lock it. This can be done using the configureCashUnit function.

- ➔ Identify the defective module (see 9.11 How to find which module is in default?)
- ➔ Modify the CashUnit.PhysicalCashUnits.Locked property to true. Call the `Bnr::ConfigureCashUnit()` method with this structure to the BNR.
- ➔ Send a reset to the BNR.

This will make the BNR change his behavior. If a recycler is locked, the banknotes assigned to this recycler will go to the cashbox. You can lock all recyclers and loader if necessary, the BNR will then become a simple banknote acceptor.

9.13 How to debug using API logs?

When integrating the BNR, it may be useful to have a look to what is really sent over the USB line. In order to do this, you have to activate first the logs of the API. Simply go to the folder where the API is installed and open “BnrCtl.ini”. Then simply turn on or off the different logs you are interested in.

- ➔ Request: The most detailed log, contain all the data send over USB to the BNR, function calls, parameters and returns from the BNR.
- ➔ API: Contains the calls made by the Host to the BNR. No parameters or callback are present here.
- ➔ Error: Contains the details of the crashes.

The details found in the request.log files can help you to confirm that you are sending the structure you created in your code and that the BNR is responding as you expect. It can help you to trace some errors, like calling a `Bnr::CashIn()` method before a `Bnr::CashInStart()` method.

10 Appendices

10.1 USB Communication pipes

As specified by the USB standard, the pipe 0 is implemented and used only for USB service. The BNR defines and uses 4 other pipes for its communications as described in the table below.

Pipe number	Type	Direction	Description
1	Bulk In	BNR to Host	BNR response to a method call
2	Bulk Out	Host to BNR	Method call for the commands (Bnr::XX and Module::XX)
3	Interrupt In	BNR to Host	Method calls for asynchronous events
4	Bulk Out	Host to BNR	Response to asynchronous events

10.2 BNR Statuses