GA973

SSP Implementation Guide

INTELLIGENCE IN VALIDATION

Innovative Technology
INTELLIGENCE IN VALIDATION

Bellis Technology
INTELLIGENCE IN VALIDATION

CONTENTS

1 Introduction 3
2 Overview Of SSP 5
- 2.1 General Information 5
- 2.2 Advantages 5
- 2.3 Requirments 5
- 2.4 Support Available 6

3 Communicating With Devices Using SSP 7
- 3.1 Packet Format 7
- 3.2 Ports and Addresses 8
- 3.3 Example 9

4 Encryption 10
- 4.1 Encryption Algorithm 10
- 4.2 Encryption Keys 10
- 4.3 Packet Format 12
- 4.4 Example 12

5 ITL Libraries 13
- 5.1 Data Structures 13
- 5.2 Initalising Libraries 15
- 5.3 Linking Functions from the Library 15
- 5.4 Setting Up The Command Structure 19
- 5.5 Constructing SSP Packets 20

6 Communicating With a Slave Device 21
- 6.1 Communication Overview 21
- 6.2 Using AN ITL Library To Start Communication 23
- 6.3 Initialisation Of The Device 23

7 Polling Devices 26
- 7.1 Poll Overview 26
- 7.2 Catching Multiple Poll Responses In One Response Data Packet 26
- 7.3 The Importance of Poll Handling 27
- 7.4 Poll Delay 27
- 7.5 Handling Events That Require More Data 27

8 ITL Devices SSP Operation 28
- 8.1 Generic Commands 28
- 8.2 Generic Responses 32
- 8.3 Bank Note Validator 33
- 8.4 NV11 44
- 8.5 SMART Payout 52
- 8.6 SMART Hopper 69

9 Commands for ITL Devices 86
- 9.1 Bank Note Validator (NV9USB, NV10USB, BV20, BV50, BV100, NV200) 86
- 9.2 NV11 92
- 9.3 SMART Payout 99
- 9.4 SMART Hopper 112

10 Updating Devices in SSP 123
- 10.1 File Structure 123
- 10.2 Process Overview 124
- 10.3 Process Details 126

11 Library Reference 132
- 11.1 ITLLib.dll 132
- 11.2 ITLSSPProc.dll 137

12 Appendix 148

Copyright Innovative Technology Ltd 2013

1 INTRODUCTION

This document provides information and resources that explain how to implement and integrate the Smiley Secure Protocol (SSP) and the encrypted version Encrypted Smiley Secure Protocol (eSSP) into a cash handling application.

This document is intended for those who will be implementing SSP/eSSP to communicate with a cash handling device.

This manual is intended for use alongside the SDKs developed by ITL. These SDKs are available in multiple programming languages and for multiple combinations of units. Please contact your local support office for more information.

WARNING

- If you do not understand any part of this manual please contact your local support office for assistance; contact details are over the page. In this way we may continue to improve our product.
- Innovative Technology Ltd has a policy of continual product improvement. As a result the products supplied may vary from the specification described here.

MAIN HEADQUARTERS	Innovative Technology Ltd Derker Street – Oldham – England - OL1 4EQ Tel: +44 161 626 9999 Fax: +44 161 620 2090 E-mail: support@innovative-technology.co.uk Web site: www.innovative-technology.co.uk

GROUP

EMAIL CONTACTS

BRAZIL
suporte@bellis-technology.com.br

CHINA
support@innovative-technology.co.uk

COLOMBIA
support@automated-transactions.net

GERMANY
support@automated-transactions.de

UNITED KINGDOM
support@innovative-technology.co.uk

SPAIN
supportes@innovative-technology.eu

UNITED STATES OF AMERICA
supportusa@bellis-technology.com

REST OF THE WORLD
support@innovative-technology.co.uk

2 OVERVIEW OF SSP

2.1 GENERAL INFORMATION

Smiley Secure Protocol (SSP) is a serial communication protocol designed by Innovative Technology LTD to address problems historically experienced by cash handling systems in gaming machines. Problems such as acceptor swapping, reprogramming acceptors and line tapping.

Since its first release in May 1998 the SSP protocol has developed and expanded to include the functionality offered by the latest generation of cash handling devices.

The interface uses a master slave communication model, the host machine is the master and the devices (Note Validator, SMART Hopper, SMART Payout) are the slaves. The devices will respond to commands sent from the host machine using a bi-directional serial transmission.

See product documentation and GA138 (eSSP Specification) for details of the hardware connections and requirements.

2.2 ADVANTAGES

SSP is an established communication protocol for cash handling devices, used a wide variety of kiosk and gaming applications worldwide.

With the encrypted layer implemented it provides secure communication between the host and the devices inside the system that cannot be manipulated externally. The security is in the key rather than relying on a cipher algorithm and as such provides substantially higher security that other cash handling protocols available.

Using SSP provides the full range of functionality provided by Innovative Technology's bank note validators, recyclers and coin handling devices.

The latest releases of datasets and firmware can be loaded into the device in SSP using the host application firmware. If the infrastructure is provided to the host machine, this can allow remote updates further increasing the security and functionality of the cash handling system.

The SSP specification is an open standard that is available for download from http://www.innovative-technology.co.uk and can be implemented by any device manufacturer without restrictions on license or royalties.

2.3 REQUIRMENTS

To communicate using SSP a bi-direction serial port or USB port capable of operating at 9600 baud is required. A 16 bit CRC is required to be calculated, this is the biggest processing overhead in the unencrypted SSP communication.

To use the encrypted layer, more processing is required to compute the encryption using 128 bit AES key. We recommend a processor that can perform AES using hardware however a processor running at least 200MHz should be capable of performing the calculation.

Copyright Innovative Technology Ltd 2013

2.4 SUPPORT AVAILABLE

Innovative Technology LTD. strives to make SSP as straightforward as possible to implement for a quick development, prototype and time to market. In addition to this comprehensive document we provide:

- Libraries (DLL for Microsoft Windows and C++ files for Linux) of methods and data structures to enable fast development. The methods provide functions like send commands and setup encryption keys. They are described more in section 0.
- Example applications demonstrating the use of the libraries and examples of polling the validator and handling the responses.

Innovative Technology LTD support engineers are available in the regional offices to assist with training and implementation of the SSP protocol; contact details are at the beginning of this manual or on the website – http://www.innovative-technology.co.uk.

3 COMMUNICATING WITH DEVICES USING SSP

3.1 PACKET FORMAT

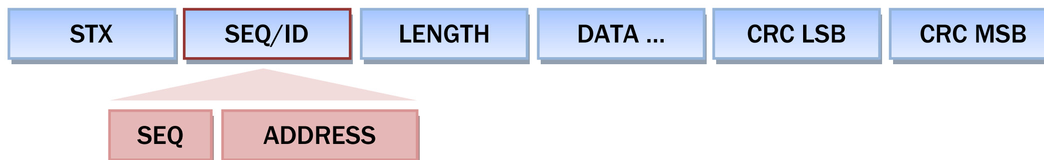
A packet is a formatted collection of data. The SSP packets are constructed using a sequence of bytes where each position in the sequence of bytes represents a field; as described below.

INFORMATION

In all documentation and examples of SSP Innovative Technology refer to the bytes in hexadecimal format (using the notation 0x0A in documentation but not in examples for clarity of reading).

Whilst in most applications the construction, encryption, decryption and parsing of packets will be handled behind the scenes by the libraries, the description here is provided for analysis of logs and full understanding of the underlying communication.

The fields that construct the SSP packet are as shown below:



STX

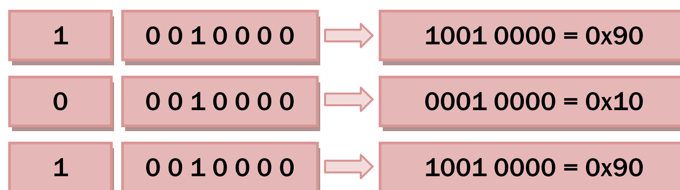
Single byte indicating the start of a packet, defined as 0x7F. If **any** other part of the packet contains 0x7F, the last step before transmission the byte should be repeated (0x7F becomes 0x7F 0x7F) to indicate it is not a STX byte; this is called byte stuffing.

SEQ/ID

A combination of two items of data: the sequence flag (MSB, bit7) and the address of the device (bit 6 to bit 0, LSB).

Each time the master sends a new packet to a device it alternates the sequence flag. If a device receives a packet with the same sequence flag as the last one, it does not execute the command but simply repeats its last reply. In a reply packet the address and sequence flag match the command packet.

For example a SMART Hopper by default has an address of 0x10 (16 decimal). When the sync bit is equal to 1 the byte sent to the Hopper is 0x90. On the next command, the sync bit is toggled, in this case 0, the byte sent would be 0x10.



Length

The number of bytes of data in the data field (including the command and all associated data), it does not include the STX, SEQ/ID, or CRC fields.

Data

The commands and/or data being sent in the packet to the device.

CRC

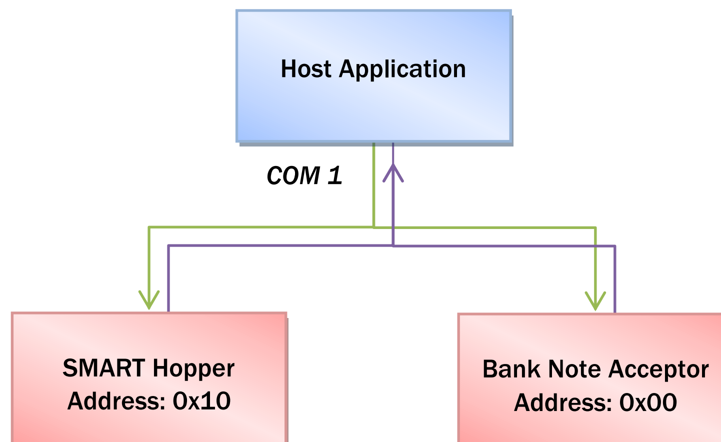
The final 2 bytes are used for a Cyclic Redundancy Check (CRC). This is provided to detect errors during transmission. The CRC is calculated using a forward CRC-16 algorithm with the polynomial ($X^{16} + X^{15} + X^2 + 1$). It is calculated on all bytes except STX and initialised using the seed 0xFFFF. The CRC is calculated before byte stuffing.

3.2 PORTS AND ADDRESSES

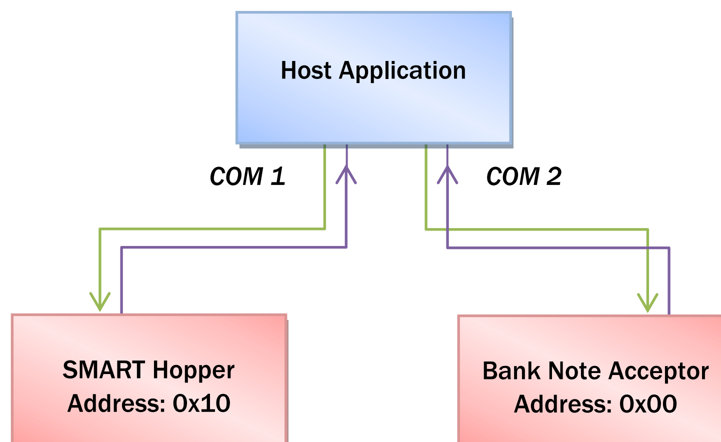
Each device using the SSP protocol has a pre-programmed address; using this address the host communicates with each device. The device will only respond to commands addressed to it. The address is echoed in the response so the host can confirm which device is responding.

There are two common models of connection for devices:

1. . Single communication port using a shared communication bus to which the RX (receive) pin and TX (transmit) pin of all devices is connected. If multiple devices are connected in this arrangement it is important that the software controls the port access carefully. This is discussed in Appendix F – Sharing Resources.



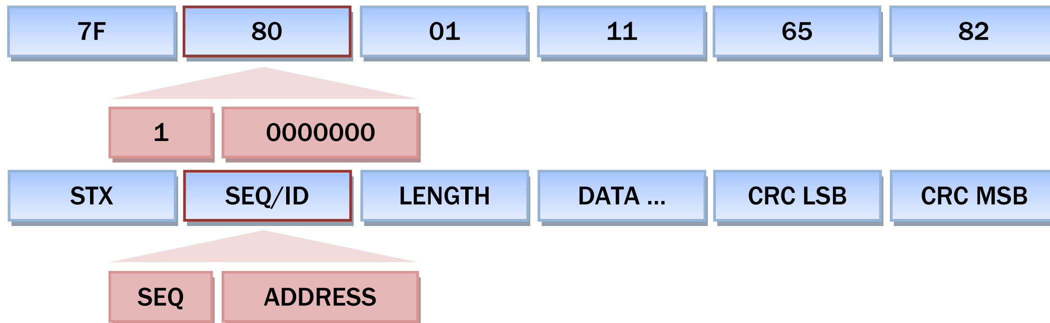
2. Two separate communication ports, with 2 separate connections. This is typical when using USB connections to multiple devices.



Copyright Innovative Technology Ltd 2013

3.3 EXAMPLE

The format of an unencrypted sync command (0x11) to a bank note validator located on address 0 would be constructed as follows:



4 ENCRYPTION

Communication with devices that support it can be encrypted using an encryption layer. This encryption ensures that all commands and responses are secure and cannot be replayed or manipulated. When using encryption, the protocol is referred to as eSSP or encrypted SSP. When an encrypted command is sent to a device that supports it, the response will be encrypted; unencrypted commands receive an unencrypted response.

Encryption is mandatory for all payout devices and optional for pay in devices. Innovative Technology LTD. recommends that encryption is used in all applications where it is within the capability of the host to perform the encryption and decryption.

There are two classes of command and response, general commands and commands involved in credit transfer. General commands may be sent with or without using the encryption layer. The device will reply using the same method. If the response contains credit information it may not be reported unless encryption is used. Credit transfer commands, a hopper payout for example, will only be accepted by the slave if received encrypted.

4.1 ENCRYPTION ALGORITHM

The encryption algorithm used in eSSP is a standard encryption method used worldwide in software for data storage and transmission called Advanced Encryption Standard (AES). In June 2003 the U.S. Government (NSA) announced that AES is secure enough to protect classified information up to the ‘top secret’ level, which is the highest security level.

eSSP implements AES with a 128-bit key. Data is encrypted in blocks of 16 bytes any unused bytes in a block should be packed with random bytes. For more details on the algorithm, please see: http://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

The SSP libraries provided by Innovative Technology LTD. contain all the algorithms required for encryption and decryption. If it is required to implement eSSP on a platform other than Microsoft Windows, Linux and compatible operating systems Innovative Technology can provide C source for the algorithms required (encryption, decryption, prime number generation, CRC calculation etc.); please contact your local support office for more details.

4.2 ENCRYPTION KEYS

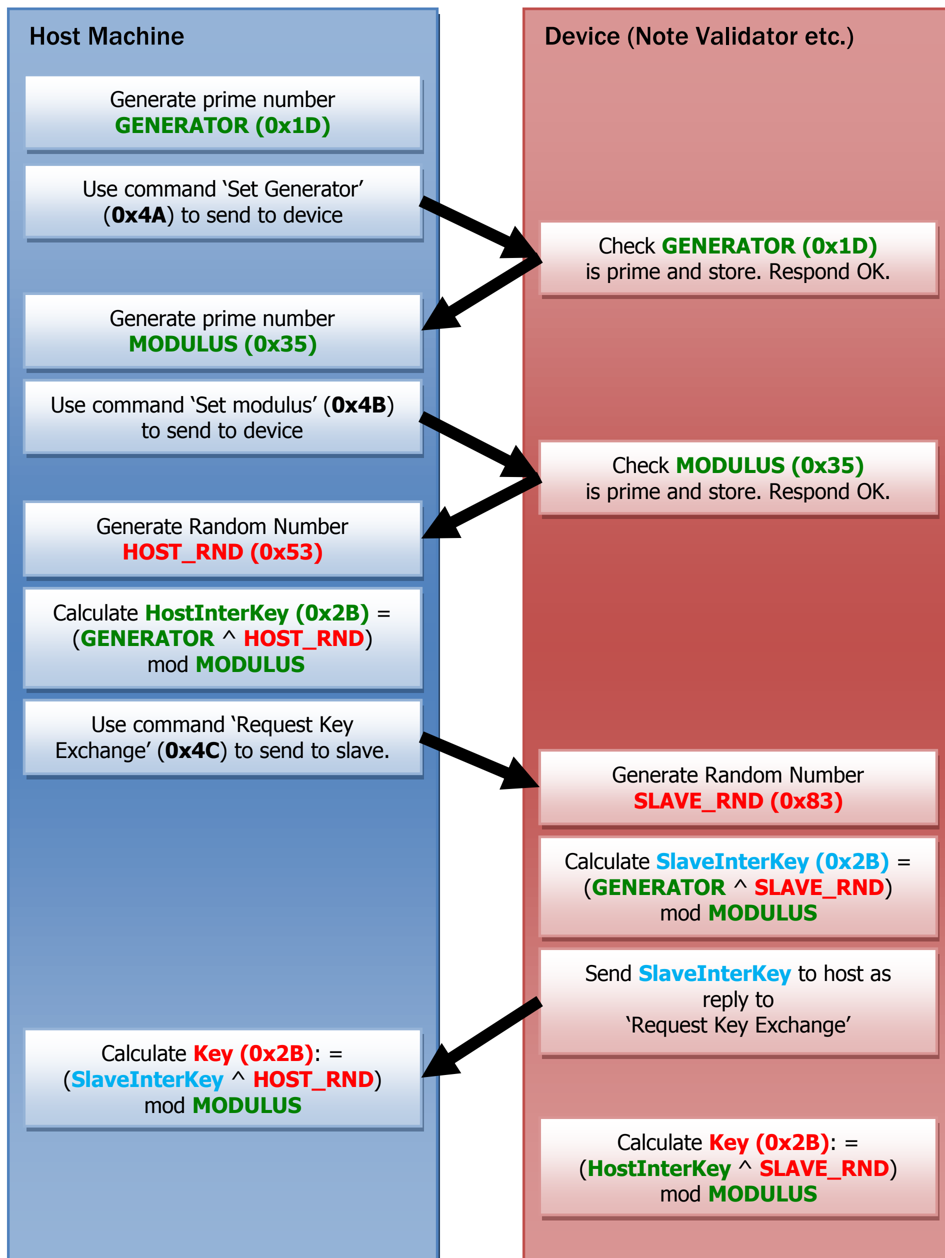
The encryption key is 128 bits long, divided into two parts. The lower 64 bits are fixed and specified by the machine manufacturer, this allows the manufacturer control which devices are used in their machines. The default for this part is “01 23 45 67 01 23 45 67” (hex bytes). The higher 64 bits are securely negotiated by the slave and host at power up, this ensures each machine and each session are using different keys.

4.2.1 KEY NEGOTIATION

The key is negotiated by the Diffie-Hellman key exchange method. This is a widely published method which allows two sides of an insecure communication link to jointly establish a shared secret key. Even if another device is monitoring the communication it is not possible to learn the key. See: http://en.wikipedia.org/wiki/Diffie-Hellman for a full description and mathematical explanation.

The implementation in eSSP is detailed below in a flowchart. Example one byte numbers have been added for clarity however in the eSSP implementation the numbers are 8 byte (64 bit).

Copyright Innovative Technology Ltd 2013



COLOUR GUIDE

Public Data Generated by Host
Public Data Generated by Slave
PRIVATE Data
Required to know and generate the key but never transmitted, keeping the key secure.

In an implementation of eSSP using the ITL libraries, the prime numbers are generated by the library. Please note that if you are generating your own prime numbers the Generator must be larger than the Modulus. If you are using the library it is only required of the

program to transmit the numbers to the device, store the response to 0x4C and to call the relevant functions.

4.3 PACKET FORMAT

The encrypted packet is wrapped inside the data field of a standard SSP packet. The encrypted section is constructed from the following fields.

STEX
Single byte indicating the start of an encrypted message; defined as 0x7E. This byte is not encrypted.

Length
The number of bytes of data in the data field (including the command and all associated data), It does not include any other fields.

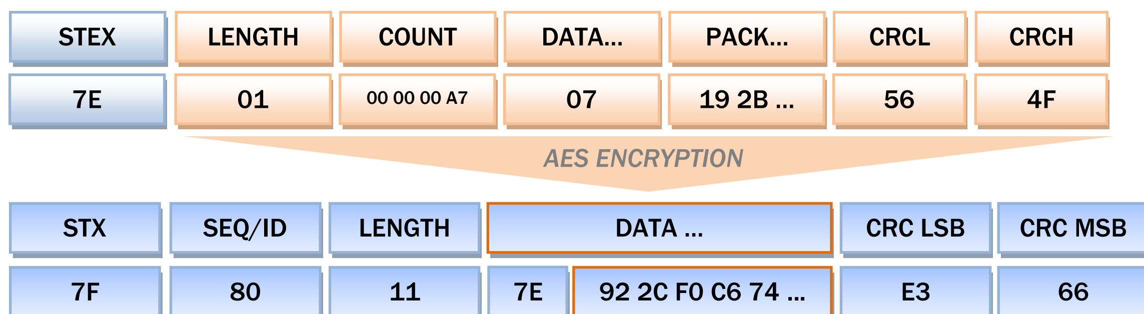
Count
A 4 byte unsigned integer representing the sequence count of encrypted packages. The packets are sequenced using this count; this is reset to 0 after a power cycle and each time the encryption keys are successfully negotiated. The count is incremented by the host and device each time they successfully encrypt and transmit a packet and each time a received packet is successfully decrypted. After a packet is successfully decrypted the COUNT in the packet should be compared with the internal COUNT, if they do not match then the packet is discarded.

Packing
Random data to make the number of bytes used in the fields LENGHT + COUNT + DATA + PACKING + CRCL + CRCH a multiple of 16 bytes. This is required for the AES algorithm.

CRCL/CRCH
As in the SSP packet, low and high byte of a forward CRC-16 algorithm using the Polynomial ($X^{16} + X^{15} + X^2 + 1$) calculated on all bytes except STEX. It is initialised using the seed 0xFFFF.

4.4 EXAMPLE

The final result of the encrypted section is then used in the data filed of a standard SSP packet as shown below, with an example of a POLL command (0x07).



The final complete packet is as follows. The encrypted bytes are highlighted in red.

7F 80 11 7E 92 2C F0 C6 74 40 D1 38 B9 17 18 4D FC 76 11 B4 E3 66

NOTE
It is not possible to decrypt the packet and examine the contents without the 128 bit key.

5 ITL LIBRARIES

Innovative Technology LTD. provide libraries (DLL for Microsoft Windows and C++ files for Linux) of methods and data structures to enable fast development. The methods provide functions like send command and setup encryption keys. They are described in full detail in section 0.

In this section the common components of the libraries are discussed, outlining how a packet is constructed and sent to the device.

> **NOTE**
> Ensure that you have the correct library for the software language and platform you are using. This will typically be "**ITLSSPProc.dll**" for C/C++ or Visual Basic 6, and "**ITLlib.dll**" for C# .NET or Visual Basic .NET.
>
> The correct library will need to be loaded at runtime if using C/C++ or Visual Basic, or referenced into your program if using .NET.

The examples contained in this section are in C++.

5.1 DATA STRUCTURES

The functions contained in the libraries are called with arguments that are references (pointers) to instances of structures that the developer will need to define in their code. These structures are detailed here for inclusion in header files.

SSP_COMMAND.

Used by the library to compile and send a packet to the validator.

```
struct SSP_COMMAND
{
	SSP_FULL_KEY Key;
	unsigned long BaudRate;
	unsigned long Timeout;
	unsigned char PortNumber;
	unsigned char SSPAddress;
	unsigned char RetryLevel;
	unsigned char EncryptionStatus;
	unsigned char CommandDataLength;
	unsigned char CommandData[255];
	unsigned char ResponseStatus;
	unsigned char ResponseDataLength;
	unsigned char ResponseData[255];
	unsigned char IgnoreError;
};
```

SSP_KEYS

Holds information about the encryption key during the eSSP key exchanging process outlined in section 4.2.1.

```
struct SSP_KEYS
{
	unsigned __int64 Generator;
	unsigned __int64 Modulus;
	unsigned __int64 HostInter;
	unsigned __int64 HostRandom;
	unsigned __int64 SlaveInterKey;
	unsigned __int64 SlaveRandom;
	unsigned __int64 KeyHost;
	unsigned __int64 KeySlave;
};
```

SSP_FULL_KEY

Hold the two components of the full 128 bit key, once it has been negotiated.

```
struct SSP_FULL_KEY
{
	unsigned long long FixedKey;
	unsigned long long EncryptKey;
};
```

SSP_COMMAND_INFO

Holds additional information about commands being sent and can be used to assist with logging communications with the validator.

```
struct SSP_COMMAND_INFO
{
	unsigned char* CommandName;
	unsigned char* LogFileName;
	unsigned char Encrypted;
	SSP_PACKET Transmit;
	SSP_PACKET Receive;
	SSP_PACKET PreEncryptTransmit;
	SSP_PACKET PreEncryptRecieve;
};
```

PORT_CONFIG

Only required when the using two or more communication ports with the same program.

```
struct PORT_CONFIG
{
        unsigned char NumberOfPorts;
        unsigned char PortID[MAX_PORTS];
        unsigned long BaudRate[MAX_PORTS];
};
```

5.2 INITALISING LIBRARIES

When loading a dynamic link library (.dll) at runtime on a Windows platform, the Windows function LoadLibrary() can be used to return a handle to the library. This handle can then be used as a parameter to the Windows function GetProcAddress() in order to obtain function addresses from the library. Both these functions require the developer to include “windows.h”.

```
// Load dll
HINSTANCE hInst = LoadLibrary("Libraries\\ITLSSPProc.dll");
if (hInst != NULL)
{
        DoWork(); // Library loaded correctly
}
```

If you are using the .Net library (ITLLib.dll) then a reference needs to be added to the project. The host software can then create an instance of the SSPComms class to interface with the device.

5.3 LINKING FUNCTIONS FROM THE LIBRARY

The functions that should be linked from the library depend on the setup of units that the developer plans to implement. When using Windows, the Windows function GetProcAddress() can be used to lookup the method names in the library and link them to a function pointer. This requires “windows.h” to be included. This step is not required for a .Net implementation.

For a single device, the only functions that will be needed are:

OPENSSPCOMPORTUSB

OpenSSPComPortUSB opens a USB communication port. The port number and parameters are defined in the instance of SSP_COMMAND structure and passed as a pointer to the method. Returns an unsigned integer.

```
typedef UINT (__stdcall* DLLFUNC1)(SSP_COMMAND* cmd);
DLLFUNC1 openSSPComPortUSB = (DLLFUNC1)GetProcAddress(hInst, "OpenSSPComPortUSB ");
```

CLOSESSPCOMPORTUSB

CloseSSPComPortUSB closes the open USB COM port. Returns an unsigned integer and requires no parameters.

```
typedef UINT (__stdcall* DLLFUNC2)(void);
DLLFUNC2 closeSSPComPortUSB = (DLLFUNC2)GetProcAddress(hInst, "CloseSSPComPortUSB");
```

Copyright Innovative Technology Ltd 2013

SSPSENDCOMMAND

SSPSendCommand constructs the packet, does any required encryption and sends to the device the command constructed in the instance of the SSP_COMMAND structure, passed as a pointer to the method. Returns an unsigned integer. As well as the pointer to the SSP_COMMAND structure instance, also takes a pointer to a SSP_COMMAND_INFO structure instance.

```
typedef UINT (__stdcall* DLLFUNC3)(SSP_COMMAND* cmd, SSP_COMMAND_INFO* sspInfo);
DLLFUNC3 sspSendCommand = (DLLFUNC3)GetProcAddress(hInst, "SSPSendCommand");
```

INITIATESSPHOSTKEYS

InitiateSSPHostKeys generates the prime numbers for generator, modulus and host_random. This returns an unsigned integer and takes a pointer to an instance of the SSP_KEYS structure along with a pointer to an instance of SSP_COMMAND structure as parameters.

```
typedef UINT (__stdcall* DLLFUNC4)(SSP_KEYS* key, SSP_COMMAND* cmd);
DLLFUNC4 initiateSSPHostKeys = (DLLFUNC4)GetProcAddress(hInst, "InitiateSSPHostKeys");
```

CREATESSPHOSTENCRYPTIONKEY

CreateSSPHostEncryptionKey is called after the device has returned the SLAVE_INTER_KEY as a response from command 0x4C. Does the mathematical functions to generate the fill 128 bit key. Returns an unsigned integer and takes a pointer to an SSP_KEYS structure as a parameter.

```
typedef UINT (__stdcall* DLLFUNC5)(SSP_KEYS* key);
DLLFUNC5 createSSPHostEncryptionKey =
                          (DLLFUNC5)GetProcAddress(hInst, "CreateSSPHostEncryptionKey");
```

5.3.2 MULTIPLE COM PORTS

The following two function descriptions are only required when the developer is planning to support multiple units in their program which are located on separate COM ports. These functions should be used **instead of** the OpenSSPComPortUSB and CloseSSPComPortUSB functions.

OPENSSPMULIPLECOMPORTS

OpenSSPMulipleComPorts opens multiple communication ports. Takes a pointer to a instance of a PORT_CONFIG structure as a parameter and returns an unsigned integer.

> **NOTE**
> This function is spelt incorrectly in the C++ library, it is not a typing error in the manual.

```
typedef UINT (__stdcall* DLLFUNC6)(PORT_CONFIG* pcnfg);
DLLFUNC6 openSSPMulipleComPorts = (DLLFUNC6)GetProcAddress(hInst, "OpenSSPMulipleComPorts");
```

CLOSESSPMULTIPLEPORTS

CloseSSPMultiplePorts closes all open COM ports. Returns an unsigned integer and takes no parameters.

```
typedef UINT (__stdcall* DLLFUNC7)(void);
DLLFUNC7 closeSSPMultiplePorts = (DLLFUNC7)GetProcAddress(hInst, "CloseSSPMultiplePorts");
```

5.3.3 VALIDATING LIBRARY LINKING

Once these function pointers have been defined and linked to the library, it is recommended to verify the process was successfully completed as shown in the example below.

```
typedef UINT (__stdcall* DLLFUNC3)(SSP_COMMAND* cmd, SSP_COMMAND_INFO* sspInfo);
DLLFUNC3 SSPSendCommand = (DLLFUNC3)GetProcAddress(hInst, "SSPSendCommand");

// Check for failure
if (SSPSendCommand == NULL)
{
        FreeLibrary(hInst);
        return false;
}
```

Copyright Innovative Technology Ltd 2013

5.4 SETTING UP THE COMMAND STRUCTURE

The command structure (SSP_COMMAND) is key to the SSP communication and contains the data, port information and additional data.

There are a set of values that need to be set before it can be passed to the libraries and any communication with the device can take place. Variables that need setting by the developer are as follows.

Variable	Description	Recommended Value
BaudRate	The speed of the communications between the unit and the host.	9600
SSPAddress	The SSP address of the device.	Varies, default for a note validator is 0x00 and a hopper is 0x10.
Timeout	The length of time in milliseconds the library will wait for a response from the device before retrying the command.	1000 (1 second)
RetryLevel	The number of times the library will retry sending a packet to the device if no response is received.	3
PortNumber	The port number of the COM port the device is connected to.	Varies based on the host machine.
EncryptionStatus	Whether the library will encrypt algorithm the packet data before it is transmitted (eSSP).	False until key is negotiated, true after.

EXAMPLE

```
SSP_COMMAND* commandStructure = new SSP_COMMAND();
commandStructure->BaudRate = 9600;
commandStructure->SSPAddress = 0x10;
commandStructure->Timeout = 1000;
commandStructure->RetryLevel = 3;
commandStructure->PortNumber = 7;
commandStructure->EncryptionStatus = false;
```

It is typical to have a separate SSP_COMMAND instance for each connected device.

Copyright Innovative Technology Ltd 2013

5.5 CONSTRUCTING SSP PACKETS

When using the ITL SSP library to send a command, the only two fields that need to modified are:

- Data
  Represented as an array of bytes, 255 bytes are available in the array.
- Length
  A single byte representing the number of bytes used in the data array for the command and associated data.

These bytes are held in the SSP_COMMAND command structure. The other fields that make up the packet are populated by library functions. If the user does not use the ITL SSP library they will be required to construct the packet in its entirety for transmission.

The following example shows the construction and transmission of a sync command (0x11) using the DLL in C++.

```
CommandStructure->CommandData[0] = 0x11;
CommandStructure->CommandDataLength= 1;
// The command is now ready to send
// (providing the command structure has been initialised)
SSPLibrary->SSPSendCommand(CommandStructure, InfoStructure);
```

6 COMMUNICATING WITH A SLAVE DEVICE

These steps provide an overview of what is required to communicate with a slave device. Specific flow charts follow in section 7, describing the events for each specific slave device. All the examples in this section are written in Windows C++, for other languages please see the Appendix.

6.1 COMMUNICATION OVERVIEW

6.1.4 SENDING A COMMAND USING THE ITL LIBRARY

A “command” refers to the data located inside the COMMAND_STRUCTURE that was initialised earlier in the program. Along with the initialisation data the developer needs to setup two other variables in order to send a command to the slave using the ITL library. These are:

- The byte array inside the command structure named CommandData.
- The byte inside the command structure named CommandDataLength.

CommandData is populated with bytes referring to the operation the device is to perform. The first element is always the byte code of the operation, for example reset is 0x01. This particular command doesn't need any other data adding to the array, so the length of the array is 1. This is set in CommandDataLength.

```
commandStructure->CommandData[0] = 0x01;
commandStructure->CommandDataLength = 0x01;
// Send command
sspLibrary->SSPSendCommand(commandStructure, infoStructure);
```

Some commands require more information to be sent along with the command byte. An example of this is the command to set the channel inhibits of the device. The byte code for this is 0x02 (stored in first position in the array). For channel inhibits an additional two bytes of information need adding to the array after the command byte. This information goes in the next 2 available slots in the array. Set the CommandDataLength variable to reflect the bytes populating CommandData.

```
// To set channel inhibits
commandStructure->CommandData[0] = 0x02; // Slot 0: command byte to set
                                         // protocol
commandStructure->CommandData[1] = 0xFF; // Slot 1: first byte of info
commandStructure->CommandData[2] = 0xFF; // Slot 2: second byte of info
commandStructure->CommandDataLength = 0x03; // Length of array
// Send command
sspLibrary->SSPSendCommand(commandStructure, infoStructure);
```

Commands can be sent to the device either in plain format or encrypted. The ITL library provides methods to simplify the key negotiation and encryption process, allowing minimal effort on the developer's behalf to implement secure communication. For more information on negotiating a secure connection with the slave device, please see section 4.2.1.

6.1.5 RETRIEVING A RESPONSE FROM A SLAVE

Once a command has been sent to the device, it will send back a response. The format of a response is very similar to that of a command. The response data is located in the command structure in the byte array ResponseData. There is also a variable named ResponseDataLength, giving the length of the response data in the array. The first byte of the response data array will be a generic response byte, an example of this is the byte 0xF0; representing the “OK” response, it indicates that the slave has received the command and is acting on it. For a full list of generic responses see the Appendix.

```
// Send command
sspLibrary->SSPSendCommand(commandStructure, infoStructure);

// Check the command was successful:
if (commandStructure->ResponseData[0] == (char)0xF0)
{
       // Command received successfully and the unit is acting on it
}
else
{
       // There was a problem with the command sent
}
```

In some cases, responses provide additional data to the generic response byte. An example of this could be the command to get the serial number of the slave (0x0C), the developer would send the command and retrieve the response in the same way as the previous command, however the length of this response stored in ResponseDataLength would be 5. The first byte will be the generic response byte, followed by 4 bytes representing the 4 byte value of the serial number.

```
int serialNumber = 0;
if (commandStructure->ResponseData[0] == (char)0xF0)
{
       char* c = new char[4];
       c[0] = commandStructure->ResponseData[1];
       c[1] = commandStructure->ResponseData[2];
       c[2] = commandStructure->ResponseData[3];
       c[3] = commandStructure->ResponseData[4];
       // Conversion function
       serialNumber = ConvertBytesToInt32(c);
       delete[] c;
}
```

6.2 USING AN ITL LIBRARY TO START COMMUNICATION

Ensure the slave is powered up and connected to the host machine. The following steps detail the process of negotiating a key with the device using the ITL libraries.

6.2.1 ESTABLISH COMMUNICATION

1. Open the COM port using the library function.
2. Disable encryption temporarily in the command structure as the first packets that need to be sent are unencrypted. This is done by setting the boolean value EncryptionStatus to false.
3. Send a synchronisation packet to the slave (0x11). This is the first of the command packets that will be sent to the slave and ensures that the slave is connected correctly and responding to commands. The host should wait for an OK (0xF0) response from Sync before progressing with the startup sequence.

6.2.2 KEY NEGOTIATION

Before sending any encrypted packets the developer needs to prepare the device to receive and decrypt them. Full code examples of this step can be found in the Appendix.

1. Call the library function “initiate SSP host keys” passing a pointer to the key structure and the command structure as parameters. The library will generate two 64 bit prime numbers, the generator and the modulus, and store them in the key structure that was passed. These prime numbers are used in the encryption algorithm.
2. The generator and modulus located in the keys structure need to be sent to the slave using the set generator (0x4A) and set modulus (0x4B) commands.
3. Send the request key exchange (0x4C) command with the host inter key, the slave intermediate key can now be retrieved from the command structure response data and stored in the key structure.
4. Call the library function “create SSP host encryption key” with a reference to the key structure passed as a parameter, this then calculates the final host key and stores it in the keys structure in the variable KeyHost.
5. In the SSP_COMMAND structure there is a variable of type SSP_FULL_KEY. This structure contains 2 variables, FixedKey and EncryptKey. The encrypt key should be set to the KeyHost variable (calculated in the last step). The fixed key for standard eSSP communications is 0x0123456701234567.

Once this is complete, the EncryptionStatus flag in the SSP_COMMAND structure can be set to true to notify the library that it should begin to encrypt packets using the key stored above.

6.3 INITIALISATION OF THE DEVICE

6.3.1 SET HOST PROTOCOL VERSION

The set host protocol version command (0x06) allows the slave device to know what protocol it should report certain events in. Later protocols can include extra commands, extra response data or different response formatting. For this reason the protocol version is set immediately after the key negotiation so no commands or data are missed.

```
// Set protocol version to 7
commandStructure->CommandData[0] = 0x06;
commandStructure->CommandData[1] = 0x07;
commandStructure->CommandDataLength = 0x02;
sspLibrary->SSPSendCommand(commandStructure, infoStructure);
```

6.3.2 SETUP REQUEST

The setup request command (0x05) can be sent to a device at any time to obtain a response containing information about the configuration of the device. This information is required for the developer to set up parts of their application. An example of this is an array to hold the value stored in each channel. There are two formats of setup request response.

- Note Validator
- Hopper

For a complete description of the setup request for a particular unit, please lookup the setup request command for that unit in section 9. A brief overview is listed below.

Note Validator Setup Request Data

- Unit type.
- Firmware version.
- Country code – deprecated protocol >= 6.
- Value multiplier – deprecated protocol >= 6.
- Number of channels.
- Channel values – deprecated protocol >= 6.
- Security of channels – deprecated.
- Real value multiplier.
- Protocol version.
- Channel country codes – protocol >=6 only.
- Extended channel values – protocol >= 6 only.

Hopper Setup Request Data

- Unit type.
- Firmware version.
- Country code - deprecated protocol >= 6.
- Protocol version.
- Number of coin types.
- Coin type values.
- Coin type country codes – protocol >= 6 only.

6.3.3 SET INHIBITS / SET COIN MECH INHIBITS

These two commands are for Note Validators and Hoppers respectively. The purpose of these commands is to control whether the unit will accept notes/coins on specified channels. When a unit is first powered up, all the channels will by default be inhibited. This is so the unit will not accept any notes/coins before the host machine is ready.

Validator

The command byte 0x02 (Note Validators) is followed by two bytes making up an “inhibit register”. If a bit is set to 1 the channel will be enabled to accept notes, 0 means the notes in the associated channel will be rejected

Hopper

The command byte 0x40 (Hoppers) is followed by two bytes making up an “inhibit register” for a coin mechanism if connected. If no coin mechanism is connected the command will return Wrong Number Of Parameters (0xF3). If a bit is set to 1 the channel will be enabled to accept coins, 0 means the coins in the associated channel will be rejected

Please see the tables below for details on “inhibit registers”.

REGISTER 1							
Channel 8	Channel 7	Channel 6	Channel 5	Channel 4	Channel 3	Channel 2	Channel 1
0	0	0	0	0	1	1	1

REGISTER 2							
Channel 16	Channel 15	Channel 14	Channel 13	Channel 12	Channel 11	Channel 10	Channel 9
0	0	0	0	0	0	0	0

The above two tables represent the two bytes following the command byte in the CommandData array. In this example only channels 1, 2 and 3 will be able to receive notes/coins.

The first byte would be 0x07 (register 1) and the second byte would be 0x00 (register 2).

```
// Enable channels 1 to 3 on Note Validator
commandStructure->CommandData[0] = 0x02;
commandStructure->CommandData[1] = 0x07; // in binary = 00000111
commandStructure->CommandData[2] = 0x00; // in binary = 00000000
commandStructure->CommandDataLength = 0x03;
sspLibrary->SSPSendCommand(commandStructure, infoStructure);
```

6.3.4 ENABLE

The final command a developer will need when initialising a device is the enable command. This command effectively “turns on” the device so it can begin receiving commands such as dispense, and acting on them. It is a single byte command of 0x0A which will return a 0xF0 (OK) response if successful.

```
// Enable
commandStructure->CommandData[0] = 0x0A;
commandStructure->CommandDataLength = 0x01;
sspLibrary->SSPSendCommand(commandStructure, infoStructure);
```

NOTE

It may be beneficial to send a poll command immediately before the Enable command in order to process any events which may be left in the poll queue. It is possible that the host may not want to Enable the device based on what it receives in this Poll response.

7 POLLING DEVICES

7.1 POLL OVERVIEW

The response to a poll command is a buffered list of bytes detailing any events that have happened since the last poll command. The developer must monitor these poll responses in order to respond to the events appropriately. The response will start with the "OK" response if the poll was successful. After this will be the event(s).

```
// Poll
commandStructure->CommandData[0] = 0x07;
commandStructure->CommandDataLength = 0x01;
sspLibrary->SSPSendCommand(commandStructure, infoStructure);

// Check Poll response if poll successful
if (commandStructure->ResponseData[0] == (char)0xF0)
{
        // Poll successful
}
```

7.2 CATCHING MULTIPLE POLL RESPONSES IN ONE RESPONSE DATA PACKET

When the slave is polled it will report a list of all the latest events that have happened. There could be multiple events in this list so it is important to iterate through the entire length of the response array (upto ResponseDataLength) and check each byte for responses and then act on each one appropriately.

```
// Poll
commandStructure->CommandData[0] = 0x07;
commandStructure->CommandDataLength = 0x01;
sspLibrary->SSPSendCommand(commandStructure, infoStructure);

// Check through each poll response if first byte is "OK"
if (commandStructure->ResponseData[0] == (char)0xF0)
{
        for (int i = 1; i < commandStructure->ResponseDataLength; i++)
        {
                // Deal with specific poll responses here
                switch (commandStructure->ResponseData[i])
                {
                        // Note stacked response
                        case (char)0xEB: break;
                        // Cashbox removed response
                        case (char)0xE3: break;
                        // Cashbox replaced response
                        case (char)0xE4: break;
                        // Credit response, this has additional data
                        case (char)0xEE:
                        {
                                unsigned char credit =
                                         commandStructure->ResponseData[i+1];
                                i++; // Now skip data in loop
                                break;
                        }
                }
        }
}
```

As shown above in the case of 0xEE, some poll responses may have additional data located in the subsequent array elements and these should be handled as follows.

- When the poll event is detected (through the switch/case statement) the specification (GA138) details how many bytes following the event are associated with it. These should be accessed using an offset from the iterator variable. For example:

```
case (char)0xEE: {
credit = commandStructure->ResponseData[i+1];
```

- Once the event has been processed the iterator should be incremented by the number of bytes associated with that event such that they are not considered as part of the switch statement. For example:

```
i++;
break;
```

7.3 THE IMPORTANCE OF POLL HANDLING

The poll command and associated responses is critical to the operation of the validator.

It is vital that all polls are handled correctly. Events such as credits (where the device has accepted and stored currency) will be passed across as a poll response. If the host misses this response, it will not register the accepted currency. This can obviously cause critical issues with totals or counters in the host machine. More importantly it could cause the host machine to not give credit for a note which has been stored by the validator.

CRITICAL NOTE

It is essential that the first poll sent to a device on startup is handled correctly as it could include events like incomplete payout with the data of the previous operation.

7.4 POLL DELAY

It is important that the delay between polls is no less than **200ms**, no greater than **1000ms**. The upper limit is set in place in order to stop the event list reported back from a poll command becoming too large, the minimum limit is to avoid the processor being interrupted too often during an operation such as a note read.

NOTE

Devices will disable if no command is received (including a poll) for 10 seconds (SMART Hopper 5 seconds), a regular poll loop will prevent this from happening.

7.5 HANDLING EVENTS THAT REQUIRE MORE DATA

If a poll event is returned and it requires more information to process the response, it is not advisable to use the same command structure to send an additional command as the response could overwrite the ResponseData that is not yet parsed.

For example, a credit event is received and the program needs to send a ‘Get Serial Number’ command to verify the serial of the unit before the credit is given to the consumer.

The recommended method of doing this is to have a set of state variables that surround the loop iterating across the response from the poll command. As each response is encountered, if further action is required a flag is set and any additional data (such as channel of credit) is stored. After the complete poll response has been parsed, a set of conditional statements check the state of the flags and perform any additional operations.

Copyright Innovative Technology Ltd 2013

8 ITL DEVICES SSP OPERATION

This section outlines the operations and recommended flow of commands (and responses) for each of the categories of Innovative Technology LTD. devices.

In the flows below, green boxes are commands sent from the host and cyan boxes indicate a response from an ITL device. The contents of the packet are shown in square brackets. For example, this demonstrates the host setting the inhibits of a validator and the device responding with the OK message.

Set Inhibits [02] [FF FF]

OK [F0]

8.1 GENERIC COMMANDS

Every peripheral using SSP to communicate must support the following commands. Any extensions to the commands or responses that are specific to each device are detailed in the relevant section.

RESET (0X01)

Poll [07]

OK [F0]

Reset [01]

OK [F0]

Close port, wait until the Hopper has had a chance to reset and open port.

Sync [11]

Sync [11]

Sync [11]

OK [F0]

Exchange keys, set protocol level, setup request, get levels and enable.

Poll [07]

OK, Slave Reset, Disabled [F0][F1][F8]

SET HOST PROTOCOL VERSION (0X06)

SSP support the notion of protocol versions. This is to allow development of the protocol in new versions of firmware with support for extra poll responses and data bytes whilst enabling existing software to not receive the extra data until it has been updated.

The protocol version is raised each time additional data is reported as a response to a command or a new poll response is included. A table is provided in the appendix detailing the firmware versions and date each protocol version was introduced for each product. Adding additional commands to the specification will not prompt a raise in the protocol version as the host will not send these commands if the software has been developed without them.

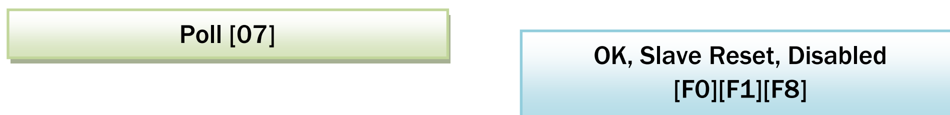
This command sets the protocol version to be used for communication and the device will respond OK (0xF0) if the protocol version is supported, if it is not supported a generic Fail (0xF8) response will be given.

Host software should be written to interface using a specific protocol version and peripherals that do not support the protocol version should be rejected by the host software until the firmware is updated to provide the correct protocol version.

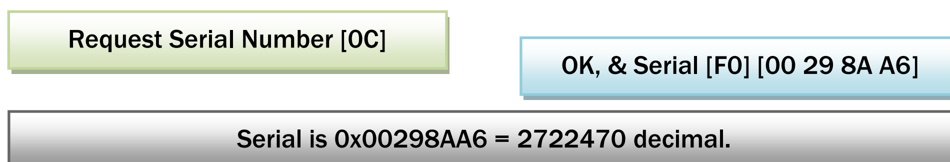
POLL (0X07)

The poll command is essential to the operation of the device and it is absolutely critical that the responses to the poll command are handled correctly at all times. The response to Poll (0x07) is a buffered list of events that have occurred since the last time a poll command was responded to.

The events that are reported in response to a poll command are specific to each device type. Please see the relevant sections for details of the events that can be returned and how to handle them.



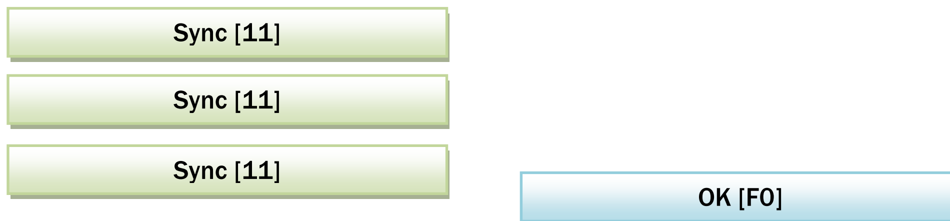
REQUEST SERIAL NUMBER (0X0C)



SYNCRONISATION (0X11)

The synchronisation (or sync) command resets the sync bit that makes up part of the seq/slave ID field. The next packet should have a seq bit of 0; this is handled transparently by the libraries and DLLs.

It is commonly used to detect when a device is present and available for communication. At the beginning of communication after opening the port, the first command should always be Sync and the host should wait for an OK (0xF0) response before any other commands are sent.



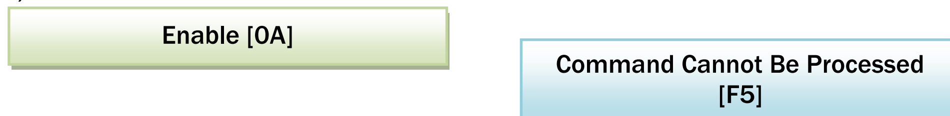
DISABLE (0X09)

Switches the device to its disabled state.

ENABLE (0X0A)

Switches the device to its disabled state.

If the device is jammed, it may return generic response Command Cannot be Processed (0xF5) instead of OK.



PROGRAM FIRMWARE/DATASET (0X0B)

This command allows the unit's firmware and note data to be updated by the host. Additional documentation covers this procedure. Contact your local support office for more details.

GET FIRMWARE VERSION (0X20)

Returns the full string detailing the current firmware installed in the device.

Get Firmware Version [20]

OK, NV02004141498000
[F0] [4E 56 30 32 30 30 34 31 34 31 34 39 38 30 30 30]

The firmware code is broken down as follows:

Returned	Reading	Meaning
NV0200	NV200	Product
414	4.14	Firmware version (Major.Minor)
1498	1498	Release ID
000	000	For internal use

GET DATASET VERSION (0X21)

Returns the full string detailing the current dataset (note/bill data) installed in the device.

Get Dataset Version [21]

OK, EUR01609
[F0] [45 55 52 30 31 36 30 39]

Returned	Reading	Meaning
EUR	Euro	Country code (ISO 4217)
01	1	Arrangement of dataset, each code has different notes or channel arrangements. See website for details.
6	NV200	Product code for dataset
09	v9	Version number, increased if notes are added, withdrawn or updated.

MANUFACTURERS EXTENSION (0X30)

This command is used exclusively for extensions to the protocol that are undocumented and generally unsupported. It should only be used if instructed specificly by Innovative Technology LTD.

Copyright Innovative Technology Ltd 2013

8.2 GENERIC RESPONSES

The following set of responses can be returned from commands. Depending on the device and response additional data can be provided along with the response to give further detail. This is detailed alongside the commands in section 9.

On receiving a response from a device, the first byte should always be checked for an OK (0xF0) response before continuing. If any other response is received it should be acted on accordingly.

OK (0XF0)

OK is the first byte returned in the response to a successful command. It does not indicate that the command has completed, just that it has been received and understood.

COMMAND NOT KNOWN (0XF2)

Returned when an invalid command is received by a peripheral. Check the firmware is up to date and the protocol level is set correctly.

WRONG NUMBER OF PARAMETERS (0XF3)

Indicates the command was received by the device but the parameters provided with the command did not match what the device was expecting.

Check the specification to ensure the arguments provided with the command were valid and that the correct protocol version is being used.

PARAMETER OUT OF RANGE (0XF4)

Indicates the command was received by the device but the parameters provided with the command were out of available range. Examples of this are providing a non-prime number to set generator command (0x4A).

COMMAND CANNOT BE PROCESSED (0XF5)

A command sent could not be processed at that time. This response can have an additional byte giving the reason the command cannot be processed. Check individual device command details for details An example of this is asking a Hopper to payout whilst it is already dispensing coins.

Check the poll response for the state of the device and retry the command when the device is enabled and not busy.

SOFTWARE ERROR (0XF6)

Reported for errors in the execution of software e.g. Divide by zero. This may also be reported if there is a problem resulting from a failed remote firmware upgrade, in this case the firmware upgrade should be redone.

FAIL (0XF8)

Used if none of the other error conditions are applicable or as detailed in command documentation. An example is setting protocol version to a number greater than that supported by the device.

KEY NOT SET (0XFA)

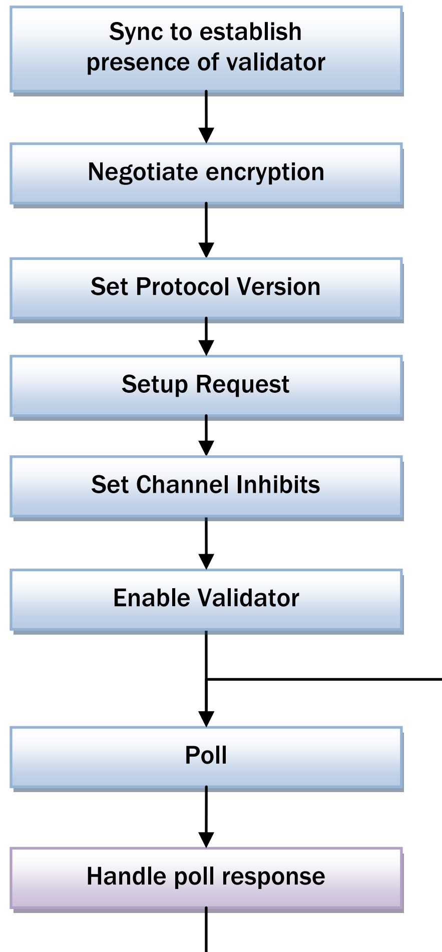
The slave is in encrypted communication mode but the encryption keys have not been negotiated.

Copyright Innovative Technology Ltd 2013

8.3 BANK NOTE VALIDATOR

OVERVIEW OF OPERATION

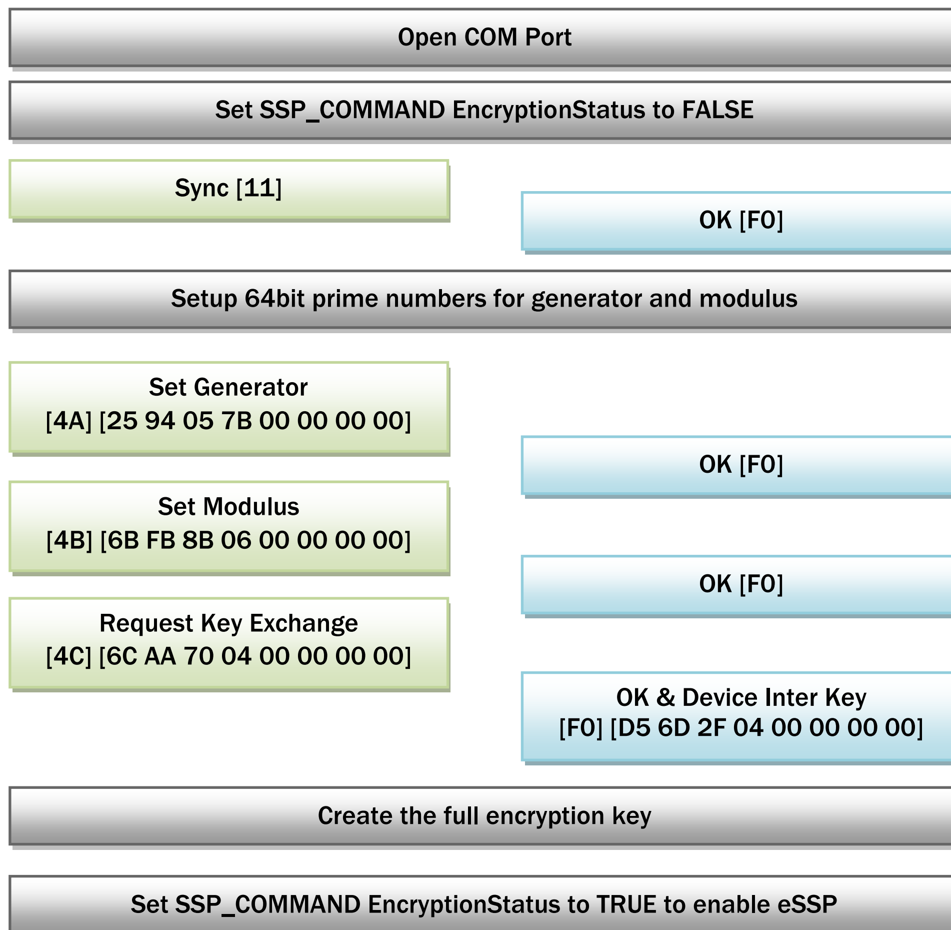
This flow outlines the blocks that make up the fundamental operation of a note validator.



NOTE

It may be beneficial to send a poll command immediately before the Enable command in order to process any events which may be left in the poll queue. It is possible that the host may not want to Enable the device based on what it receives in this Poll response.

SYNC AND KEY NEGOTIATION



The only possible response to a synchronisation command is OK (0xF0).

If a prime number is not provided as the generator or modulus the device will respond with Parameter Out Of Range (0xF4). If it was not possible to create the key, the device will respond with Fail (0xF8).

If the wrong key is used to encrypt a command, the device will respond with Key Not Set (0xFA). This will also be the response to all standard commands if encryption is forced on the unit and a key has not yet been exchanged.

Copyright Innovative Technology Ltd 2013

SETUP AND ENABLE VALIDATOR

This will enable acceptance of all notes in the dataset.

Host Protocol Version [06] [06]

OK [F0]

Setup Request [05]

OK & Setup Data [F0]
[00 30 33 33 35 45 55 52
00 00 01 04 05 0A 14 32
02 02 02 02 00 00 64 06
45 55 52 45 55 52
45 55 52 45 55 52
05 00 00 00
0A 00 00 00
14 00 00 00
32 00 00 00]

Parse setup request

00 = Unit Type (Note Validator)
30 33 33 33 = Firmware (3.33)
45 55 52 = Country Code (EUR)
00 00 01 = Value Multiplier (1)
04 = Number of channels (4)
05 0A 14 32 = Channel Value for older protocol version – ignore for v6.
02 02 02 02 = Channel Security for older protocol version – ignore for v6

00 00 64 = Real Value Multiplier (100)
06 = Protocol Version(6)
45 55 52 (repeated x 4) = Currency Code for each channel (EUR)
05 00 00 00 = Value of Ch1 (5)
0A 00 00 00 = Value of Ch2 (10)
14 00 00 00 = Value of Ch3 (20)
32 00 00 00 = Value of Ch4 (50)

Request Serial Number [0C]

OK, & Serial [F0] [00 29 8A A6]

Store serial number for later checks

Poll [07]

OK, Reset & Disabled [F0] [F1 E8]

If poll response is anything other than F1 (reset), E8 (disabled) & F0 (OK)
Handle poll response before enabling.

Set Inhibits [02] [FF FF]

OK [F0]

Enable [0A]

OK [F0]

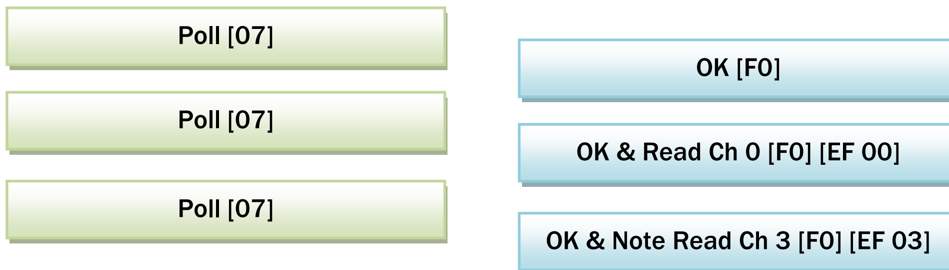
Validator now able to accept notes

Software should poll device for events

If the value provided with Set Host Protocol version is not supported by the device, the device will respond Fail (0xF8). If it responds with OK (0xF0) the protocol version is supported and will be used on the device. After startup (or a reset), it is recommended that the protocol level is always set as the first command sent after the key exchange has taken place. This will ensure no events are missed. If a device does not support the protocol version the host expects to use it is a critical error and the host should reject the device until the firmware is updated or the device changed.

If not enough arguments are passed with Set Inhibits, it will respond with Wrong Number Of Parameters (0xF3). If OK (0xF0) is returned, the inhibit mask will be used.

ACCEPT NOTE



Next command directs how the note is handled:
0x07 – Poll – Accept Note
0x18 – Hold – Keep the note in escrow position (for 5 seconds longer)
0x08 – Reject – Return the note to the bezel

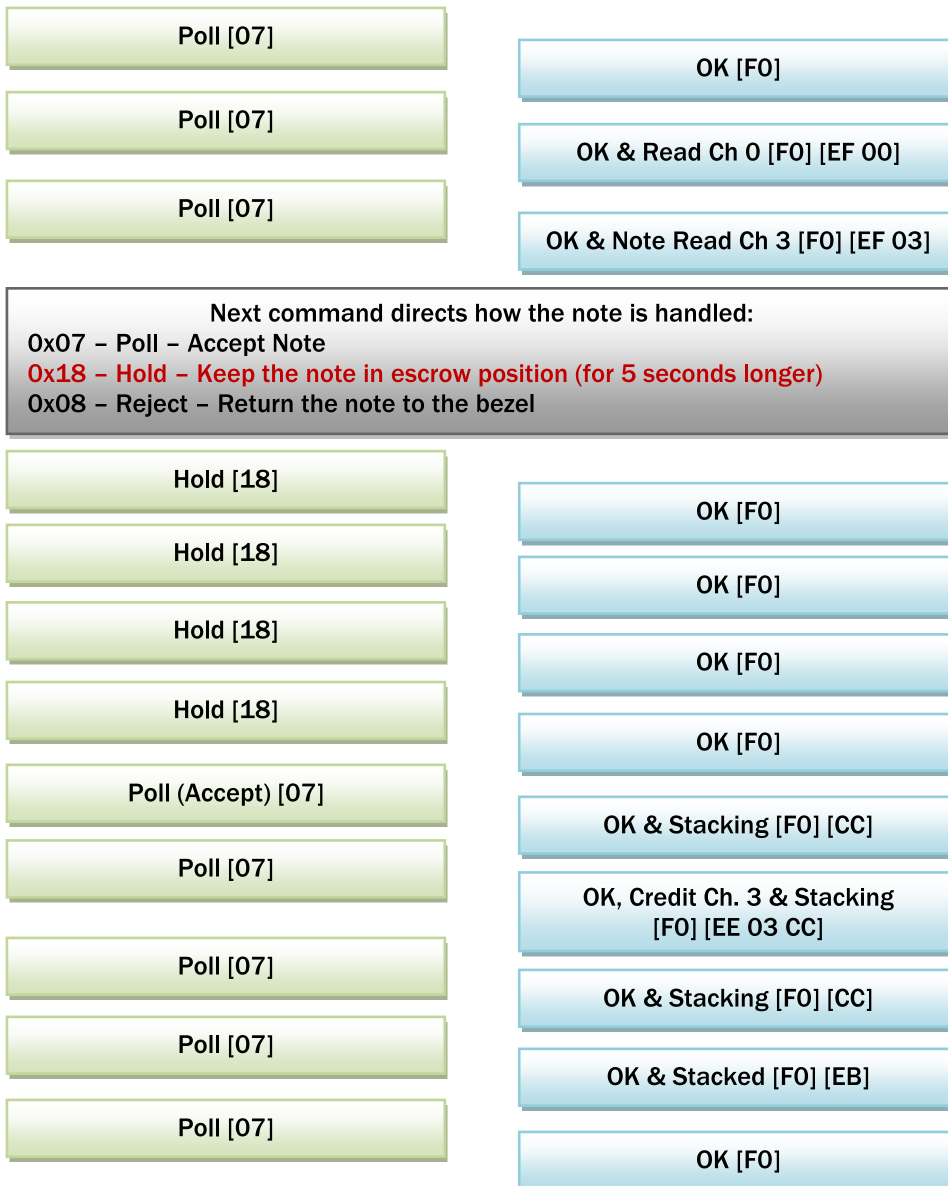
The note is now secure, it cannot be returned to the customer

For additional security, the host could request the serial number of the validator and compare it with the serial stored on startup before giving credit.

Copyright Innovative Technology Ltd 2013

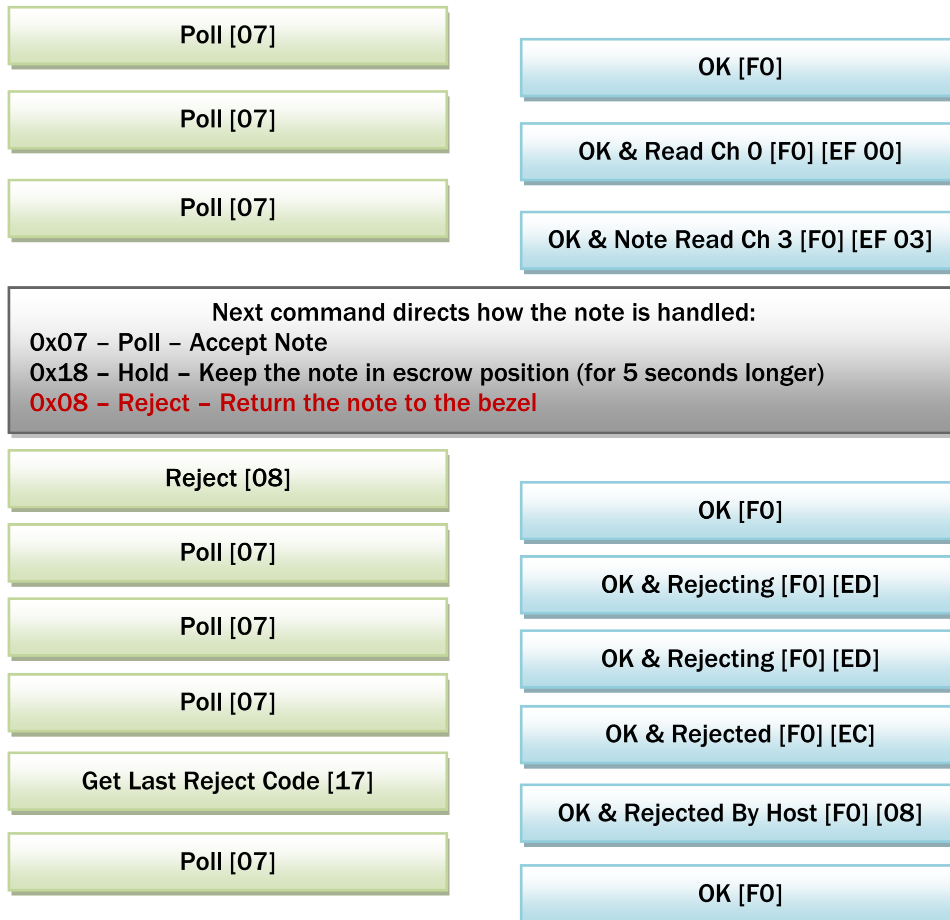
HOLD NOTE

If no command is sent to the validator for 5 seconds, the validator will return any note in the note path and inhibit itself as a security mechanism. This includes when a note is in the escrow position waiting for the host to accept or reject the note. To allow the note to remain in the escrow position longer than 5 seconds, send the Hold command (0x18) instead of Poll (0x07).



If there is not a note in the escrow position and the host sends Hold command (0x18) the device will respond with Command Cannot Be Processed (0xF5).

REJECT NOTE



If there is not a note in the escrow position and the host sends Reject command (0x08) the device will respond with Command Cannot Be Processed (0xF5).

All reject codes are detailed in Appendix.

VALIDATOR REJECTS NOTE

Host	Validator
Poll [07]	OK [F0]
Poll [07]	OK & Read Ch 0 [F0] [EF 00]
Poll [07]	OK & Read Ch 0 [F0] [EF 00]
Poll (Accept Note) [07]	OK & Rejecting [F0] [ED]
Poll [07]	OK & Rejecting [F0] [ED]
Poll [07]	OK & Rejected [F0] [EC]
Get Last Reject Code [17]	OK & Validation Fail [F0] [04]
Poll [07]	OK [F0]

SAFE JAM

The note is past the point it cannot be retrieved by the customer however there has been a jam and the process could not complete.

It is usually safe to give the credit in this case, however by default no credit event (0xEE) is reported.

Poll [07]

OK [F0]

Poll [07]

OK & Read Ch 0 [F0] [EF 00]

Poll [07]

OK & Note Read Ch 3 [F0] [EF 03]

Poll (Accept Note) [07]

OK [F0]

Poll [07]

OK & Stacking [F0] [CC]

Poll [07]

OK, Safe Jam & Disabled
[F0] [EA E8]

Poll [07]

OK, Safe Jam & Disabled
[F0] [EA E8]

Remove power, separate validator and clear jam.
Reconnect power and redo sync, key exchange and initialisation.

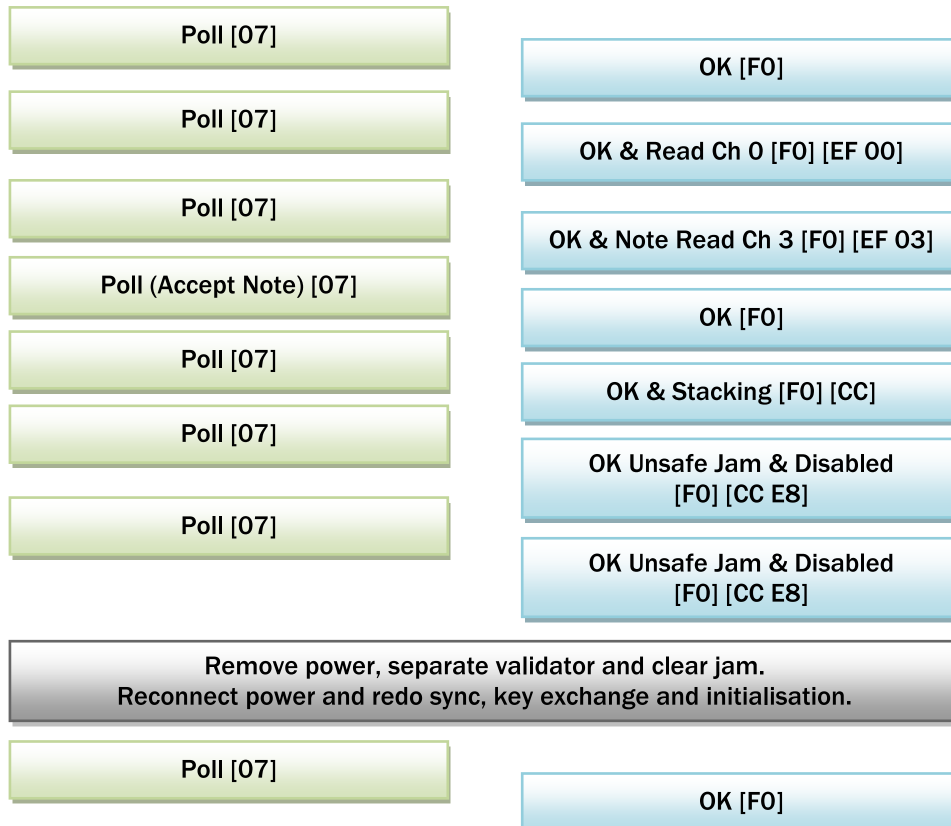
Poll [07]

OK [F0]

UNSAFE JAM

An unsafe jam is where the note is not securely in the cashbox and is jammed in the note path. It can happen at any time during the reading process, the example below is just one case, the events might not necessarily be in this order if it gets jammed sooner or later.

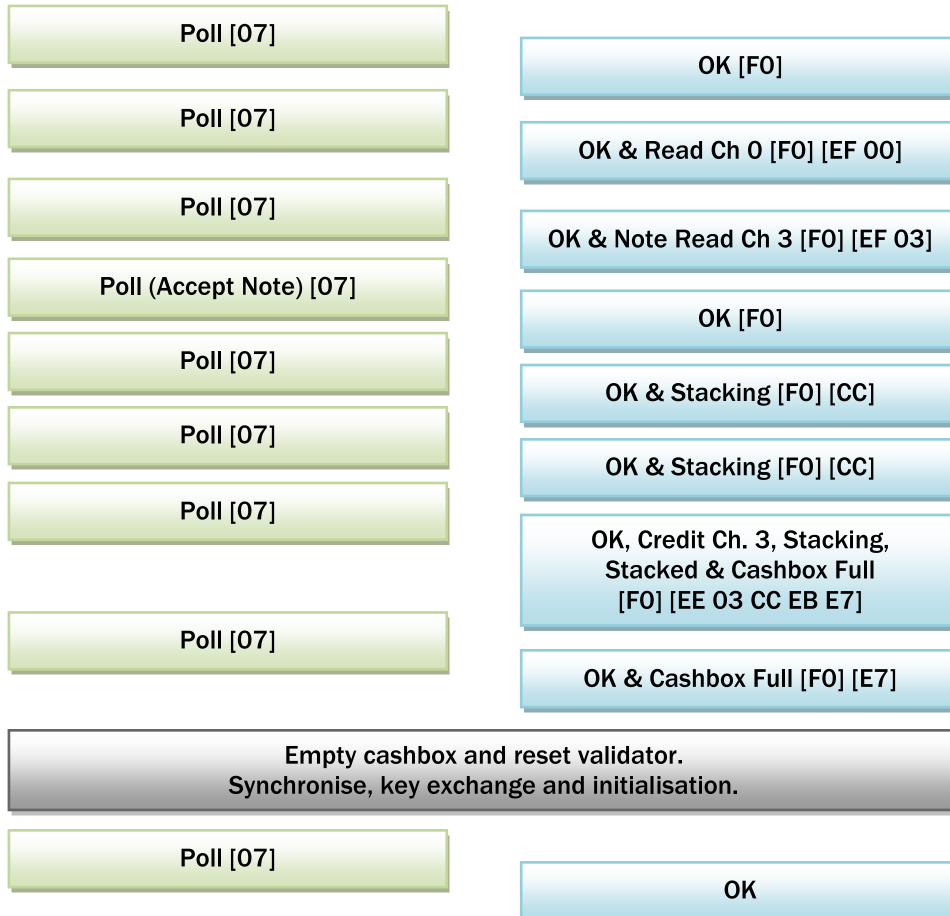
It the case of unsafe jam it is not usually advisable to give credit as it is not guaranteed the note is secure and cannot be retrieved.



STACKER FULL

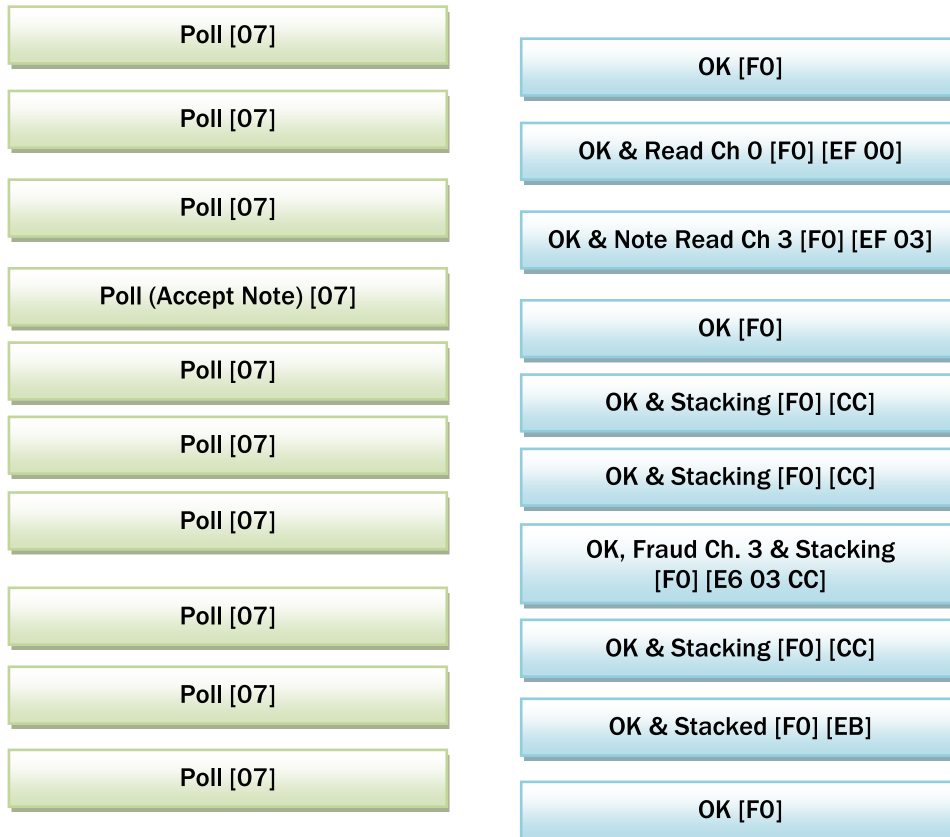
Returned as part of the stacking process during note reads when the cashbox is full.

The validator will not accept any more notes until the cashbox has been emptied and validator reset.



FRAUD ATTEMPT

During the note transport and stacking process if the validator detects a potential fraud event no credit event will be given, instead 0xEB will be reported with the channel that was being stacked at the time.



When a fraud attempt is seen as part of the note reading process it is usually because the validator has detected something is not as it should be. It is recommended that they are always logged as part of the host security log.

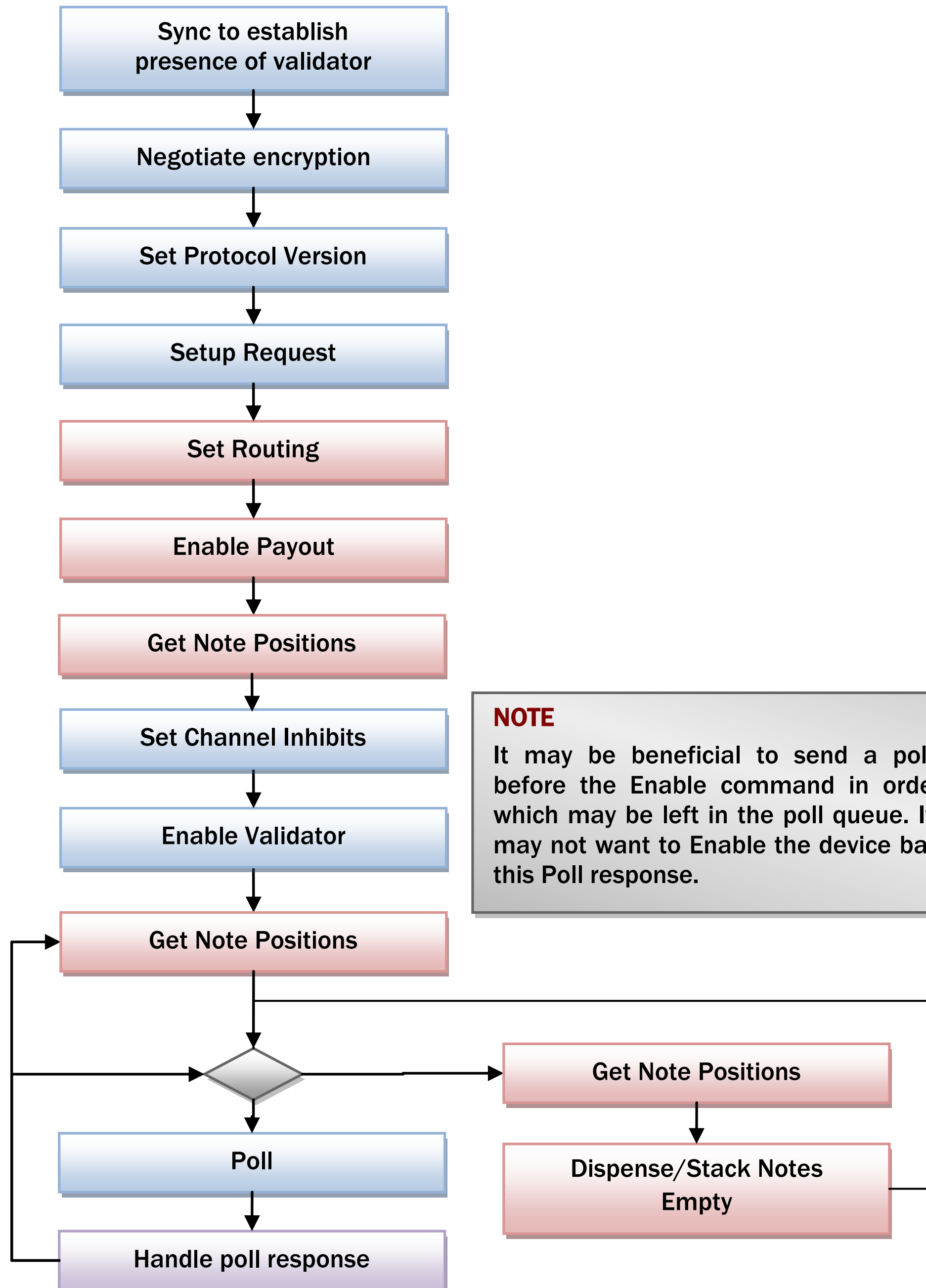
A single fraud attempt can usually be ignored as an anomaly and the credit given however if multiple fraud attempts are seen in a short space of time then the validator should be disabled until an attendant or service personnel can investigate; either:

- A manipulation is being performed on the validator and it should be disabled to prevent any further manipulation. If an attendant is on-site they should be called to investigate.
- The validator is faulty and reporting false fraud attempts. In this case disable the validator to prevent further alerts and errors. Contact your nearest authorised service centre.

8.4 NV11

OVERVIEW OF OPERATION

This flow outlines the blocks that make up the fundamental operation of the NV11.
The blocks and responses detailed in the bill validator section are common to the NV11 as well. The processed detailed below are in addition and relate to the recycling of notes.



NOTE
It may be beneficial to send a poll command immediately before the Enable command in order to process any events which may be left in the poll queue. It is possible that the host may not want to Enable the device based on what it receives in this Poll response.

ROUTING NOTES

The available notes can be retrieved using the Setup Request command during the initial setup. It is advisable to set the routing of all the notes at startup so the validator is in a known state.

In this case we have a Euro dataset programmed in the NV11:
Channel 1 = EUR 5
Channel 2 = EUR 10
Channel 3 = EUR 20
Channel 4 = EUR 50

The example below routes the EUR5 & EUR10 notes to the recycler and the EUR20 & EUR50 notes to the stacker.

Routing can be to recycler for storage to be paid/stacked later (route = 0) or to cashbox/stacker (route = 1).

Host	Device
Set Routing, Recycler, 5.00 EUR [3B] [00] [F4 01 00 00 45 55 52]	OK [F0]
Set Routing, Recycler, 10.00 EUR [3B] [00] [E8 03 00 00 45 55 52]	OK [F0]
Set Routing, Stacker, 20.00 EUR [3B] [01] [D0 07 00 00 45 55 52]	OK [F0]
Set Routing, Stacker, 50.00 EUR [3B] [01] [88 13 00 00 45 55 52]	OK [F0]

- If the Note Float is not detected as connected a Command Cannot Be Processed (0xF5) will be returned.
- If the value and currency passed is not in the dataset Parameter Out Of Range (0xF4) will be returned.
- If the command was sent unencrypted then Parameter Out Of Range (0xF4) will be returned.

GET ROUTING

The routing of each note can be confirmed using the get routing command.

Host	Device
Get Routing, 5.00 EUR [3B] [F4 01 00 00 45 55 52]	OK, Recycler [F0] [00]

- If the Note Float is not detected as connected a Command Cannot Be Processed (0xF5) will be returned.
- If the value and currency passed is not in the dataset Parameter Out Of Range (0xF4) will be returned.
- If the command was sent unencrypted then Parameter Out Of Range (0xF4) will be returned.

ENABLE PAYOUT

Enables the recycler unit for payout, stacking and storage.

Options for the configuration of the payout can be set as it is enabled. These options are set as a bit register. Currently only one option is available which is to receive the value of the note stored with the stored poll response. This is enabled by setting the Least Significant Bit to 1.

Enable Payout, Options [5C] [01]

OK [F0]

- If 0xF0 is returned, the recycler is enabled.
- If 0xF5 (command cannot be processed) is returned, an error code will follow. See the relevant command in section 9.2 for more details on these codes.
    - 0x01 No Note Float connected
    - 0x02 Invalid Currency
    - 0x03 Device busy
    - 0x04 Empty only
    - 0x05 Note float device error
- If the command was sent unencrypted then Parameter Out Of Range (0xF4) will be returned.

DISABLE PAYOUT

Disables the recycler unit from payout, stacking and storage.

All inserted notes will be stacked upon acceptance.

Disable Payout [5B]

OK [F0]

- If 0xF5 (command cannot be processed) is returned, an error code will follow. See the relevant command in section 9.2 for more details on these codes.
    - 0x01 No Note Float connected
    - 0x03 Device busy
- If the command was sent unencrypted then Parameter Out Of Range (0xF4) will be returned.

GET NOTE POSITIONS

Used to get the value of each of the notes in the recycler.

This command should be used

- During setup to know what is stored in the recycler and available for payout.
- Before any stack/dispense commands are sent to confirm the note to be dispensed.
- After any storage procedure to obtain an updated list of notes in the recycler.

Get Note Positions [41]

OK, 3 Notes,
5.00,
10.00,
10.00.
[F0] [03]
[F4 01 00 00]
[E8 03 00 00]
[E8 03 00 00]

Note: The first available note is represented by the LAST note listed in the response to this command. In this case a EUR10 note would be dispensed, followed by another EUR10 and finally an EUR5 note.

Note: When using a mixed currency dataset, it is advised that channel number reporting type is used – see command 0x45. This is due to only the value being reported and not the currency. This is illustrated below.

Set Value Reporting Type,
By Channel [45] [01]

OK [F0]

Get Note Positions [41]

OK, 3 Notes,
Ch1, Ch2, Ch2
[F0] [03]
[01] [02] [02]

- If 0xF5 (command cannot be processed) is returned, an error code will follow. See the relevant command in section 9.2 for more details on these codes.
  - 0x01 No Note Float connected or Note Float error
  - 0x03 Device busy

DISPENSING NOTES

* One Note in this context denotes that the data for one note follows, for parsing purposes. It does not directly denote the number of notes being dispensed.

Poll [07]

OK [F0]

Get Note Positions [41]

OK, 2 Notes,
10.00, 5.00,
[F0] [02]
[E8 03 00 00] [F4 01 00 00]

Dispense Note [42]

OK [F0]

Poll [07]

OK, Dispensing, 1 Note*,
0EUR, Disabled [F0]
[DA] [01] [00 00 00 00 45 55 52]
[E8]

Poll [07]

OK, Disabled, Dispensing,
1 Note*, 0EUR [F0] [E8]
[DA] [01] [00 00 00 00 45 55 52]

The value will be 0 until the note has passed out of the Note float
and into a payable position in the validator.

Poll [07]

OK, Disabled, Dispensing,
1 Note*, 500 EUR
[F0] [E8]
[DA] [01] [F4 01 00 00 45 55 52]

Poll [07]

OK, Disabled,
Dispensing, 1 Note*, 500 EUR
Note In Bezel, 1 Note*, 500 EUR
[F0] [E8]
[DA] [01] [F4 01 00 00 45 55 52]
[CE] [01] [F4 01 00 00 45 55 52]

Poll [07]

OK, Disabled,
Note In Bezel, 1 Note*, 500 EUR
[F0] [E8]
[CE] [01] [F4 01 00 00 45 55 52]

Persistent until note is removed from bezel by customer.

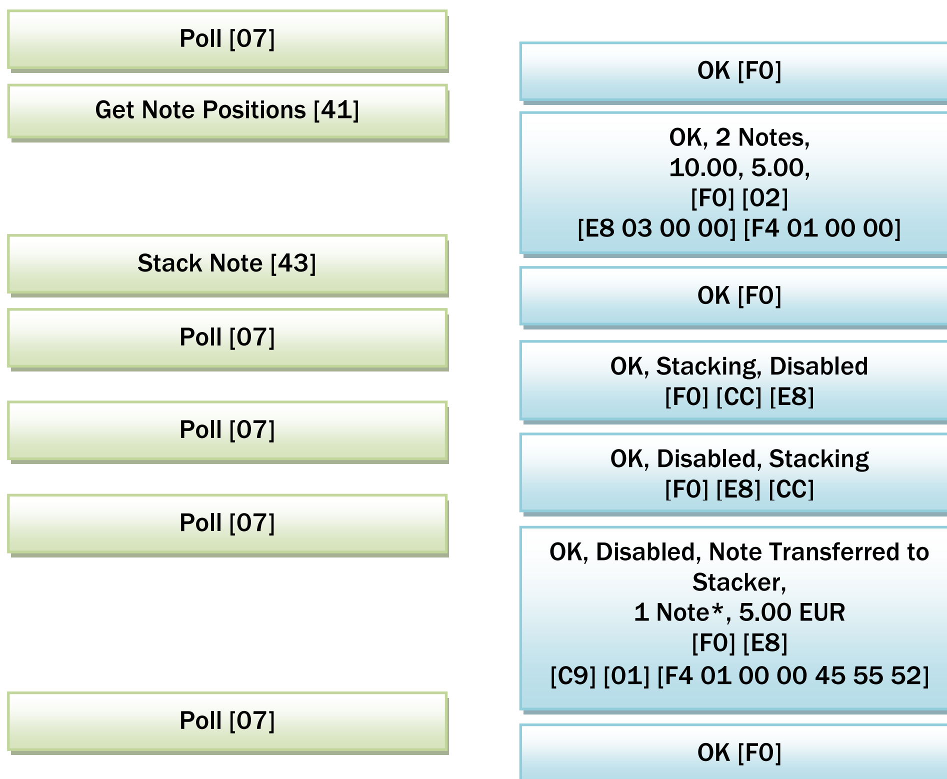
Poll [07]

OK, Disabled,
Dispensed, 1 Note*, 500 EUR
[F0] [E8]
[D2] [01] [F4 01 00 00 45 55 52]

The device will respond to Dispense Note command (0x42) with OK (0xF0) if there are no problems. Alternatively the following responses could be received:

- 0xF5 (command cannot be processed) is returned, an error code will follow. See the relevant command in section 9.2 for more details on these codes.
    - 0x01
        - No Note Float connected
        - The validator has a dataset installed that does not match that of the notes stored in the recycler.
        - Note Float error
    - 0x02 Note Float Empty
    - 0x03 Payout Busy
    - 0x04 Note Float Disabled

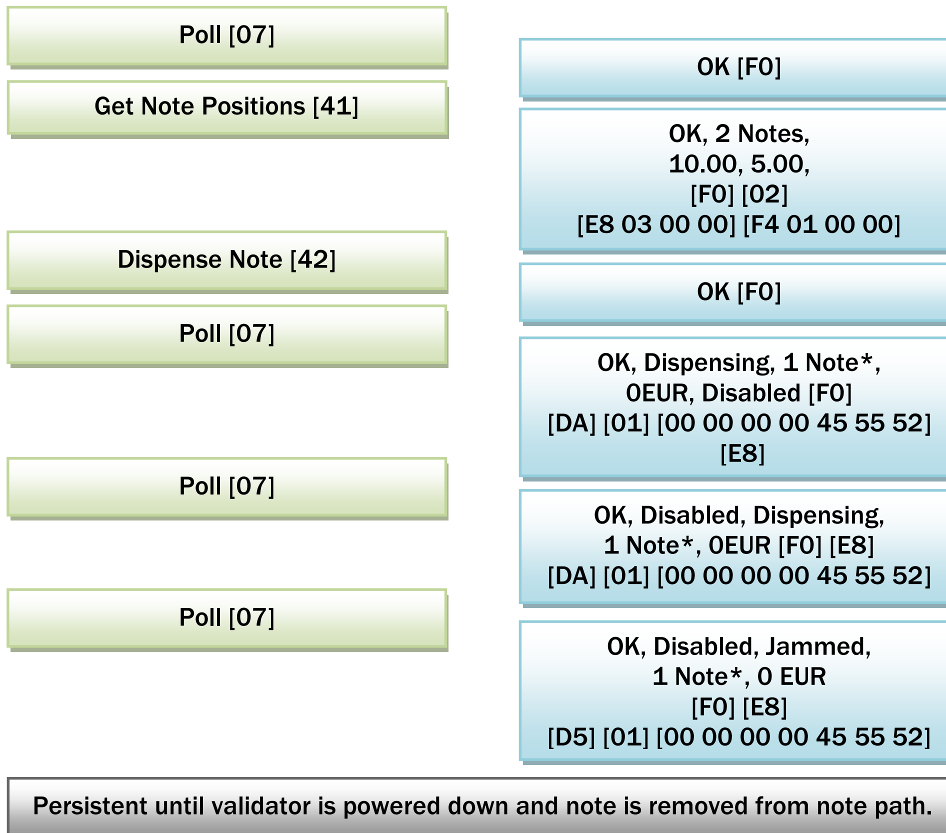
STACKING NOTES



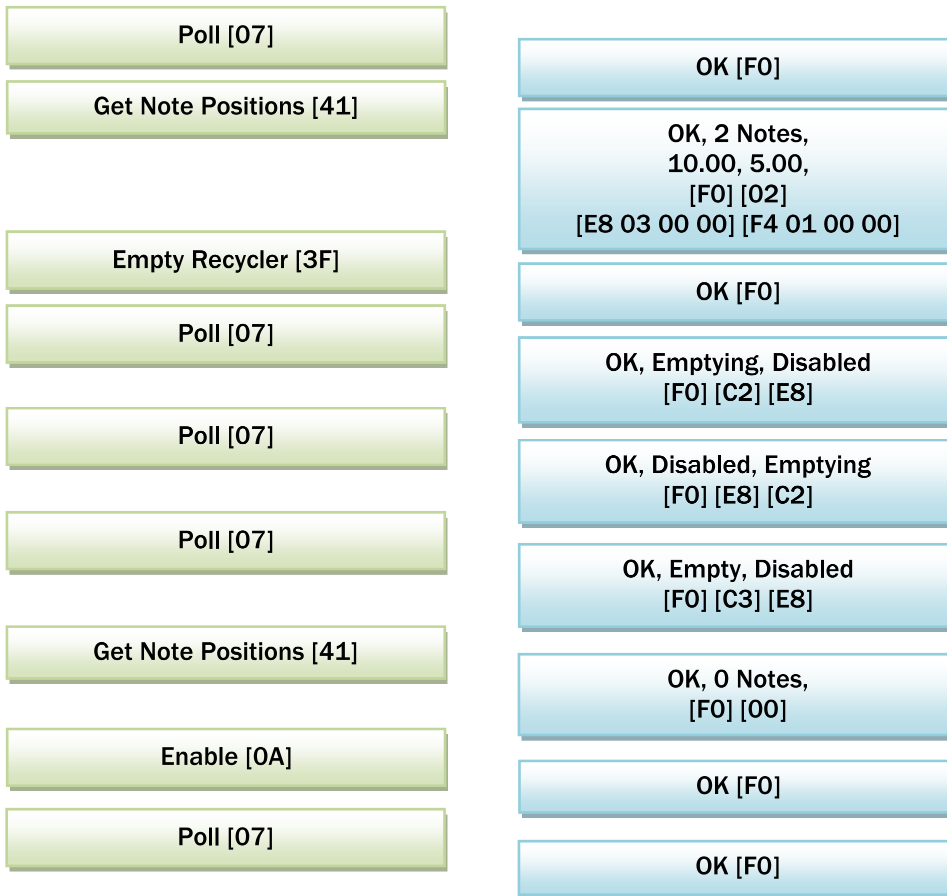
The device will respond to Stack Note command (0x43) with OK (0xF0) if there are no problems. Alternatively the following responses could be received:

- 0xF5 (command cannot be processed) is returned, an error code will follow. See the relevant command in section 9.2 for more details on these codes.
    - 0x01
        - No Note Float connected
        - The validator has a dataset installed that does not match that of the notes stored in the recycler.
        - Note Float error
    - 0x02 Note Float Empty
    - 0x03 Payout Busy
    - 0x04 Note Float Disabled

NOTE TRANSPORT ERROR DURING DISPENSE



EMPTY RECYCLER



The device will respond to Empty command (0x3F) with OK (0xF0) if there are no problems. Alternatively the following responses could be received:

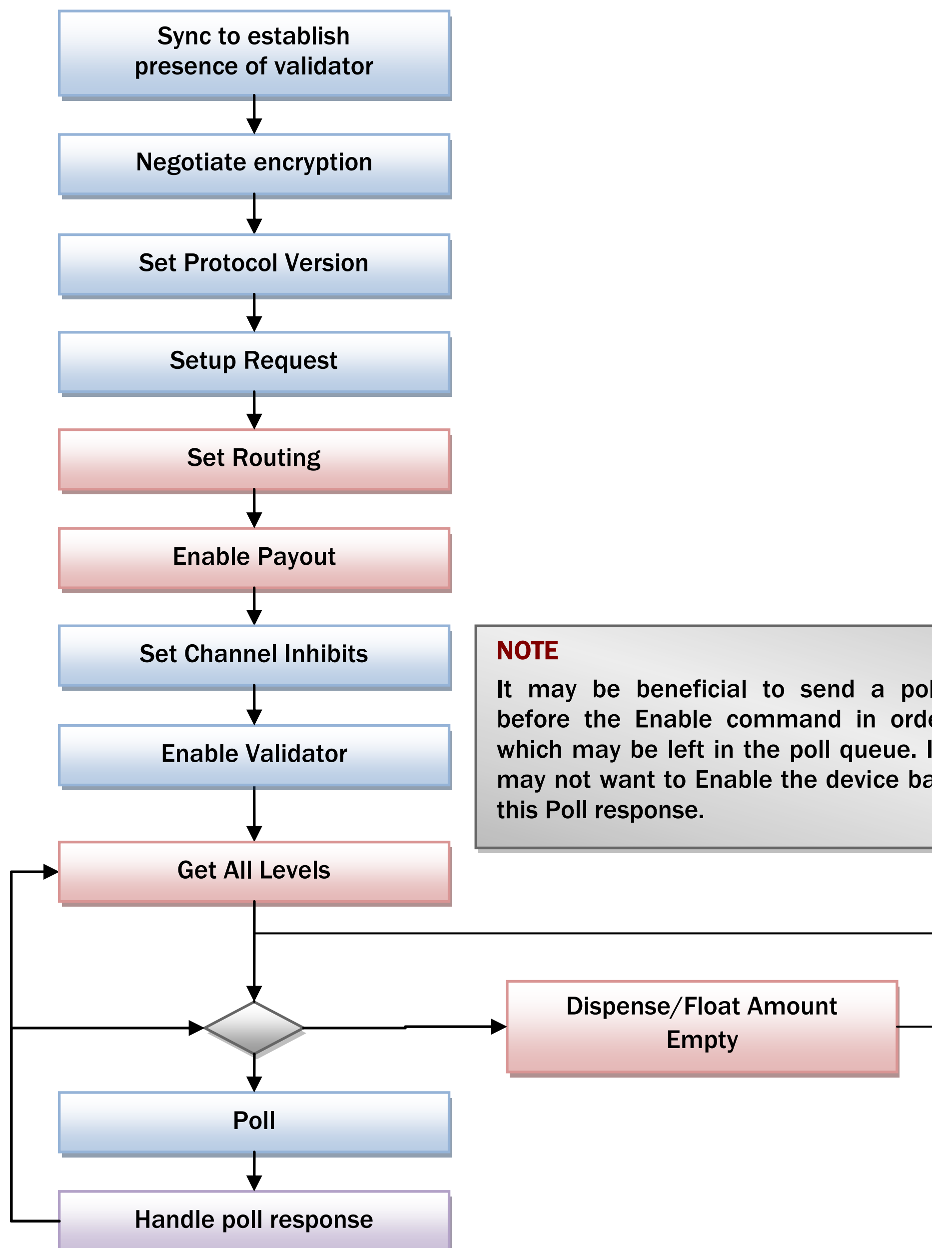
- 0xF5 (command cannot be processed) is returned, an error code will follow. See the relevant command in section 9.2 for more details on these codes.
    - 0x01
        - No Note Float connected
        - Note Float error
    - 0x02 Note Float Empty
    - 0x03 Payout Busy
    - 0x04 Note Float Disabled
- If the command was sent unencrypted then Parameter Out Of Range (0xF4) will be returned.

8.5 SMART PAYOUT

OVERVIEW OF OPERATION

This flow outlines the blocks that make up the fundamental operation of a SMART Payout.

The blocks and responses detailed in the bill validator section are common to the SMART Payout as well. The processed detailed below are in addition and relate to the recycling of notes in the SMART Payout.



NOTE

It may be beneficial to send a poll command immediately before the Enable command in order to process any events which may be left in the poll queue. It is possible that the host may not want to Enable the device based on what it receives in this Poll response.

ROUTING NOTES

The available notes can be retrieved using the Setup Request command during the initial setup. It is advisable to set the routing of all the notes at startup so the validator is in a known state.

In this case we have a Euro dataset programmed in the NV200:
Channel 1 = EUR 5
Channel 2 = EUR 10
Channel 3 = EUR 20
Channel 4 = EUR 50

The example below routes the EUR5, EUR10 & EUR20 notes to the recycler and the EUR50 notes to the stacker.

Routing can be to recycler for storage to be paid/stacked later (route = 0)
or to cashbox/stacker (route = 1).

Host	Validator
Set Routing, Recycler, 5.00 EUR [3B] [00] [F4 01 00 00 45 55 52]	OK [F0]
Set Routing, Recycler, 10.00 EUR [3B] [00] [E8 03 00 00 45 55 52]	OK [F0]
Set Routing, Recycler, 20.00 EUR [3B] [00] [D0 07 00 00 45 55 52]	OK [F0]
Set Routing, Stacker, 50.00 EUR [3B] [01] [88 13 00 00 45 55 52]	OK [F0]

- If the Payout is not detected as connected a Command Cannot Be Processed (0xF5) will be returned.
- If the value and currency passed is not in the dataset Parameter Out Of Range (0xF4) will be returned.
- If the command was sent unencrypted then Parameter Out Of Range (0xF4) will be returned.

GET ROUTING

The routing of each note can be confirmed using the get routing command.

Host	Validator
Get Routing, 5.00 EUR [3B] [F4 01 00 00 45 55 52]	OK, Recycler [F0] [00]

- If the Payout is not detected as connected a Command Cannot Be Processed (0xF5) will be returned.
- If the value and currency passed is not in the dataset Parameter Out Of Range (0xF4) will be returned.
- If the command was sent unencrypted then Parameter Out Of Range (0xF4) will be returned.

ENABLE PAYOUT

Enables the recycler unit for payout, stacking and storage.

Options for the configuration of the payout can be set as it is enabled. These options are set as a bit register. Currently only one option is available which is to receive the value of the note stored with the stored poll response. This is enabled by setting the Least Significant Bit to 1.

Enable Payout, Options [5C] [01]

OK [F0]

- If 0xF0 is returned, the recycler is enabled.
- If 0xF5 (command cannot be processed) is returned, an error code will follow. See the relevant command in section 9.3 for more details on these codes.
    - 0x01 No Payout connected
    - 0x02 Invalid Currency
    - 0x03 Device busy
- If the command was sent unencrypted then Parameter Out Of Range (0xF4) will be returned.

DISABLE PAYOUT

Disables the recycler unit from payout, stacking and storage.

All inserted notes will be stacked upon acceptance.

Disable Payout [5B]

OK [F0]

- If 0xF5 (command cannot be processed) is returned, an error code will follow. See the relevant command in section 9.3 for more details on these codes.
    - 0x01 No Note Float connected
    - 0x03 Device busy
- If the command was sent unencrypted then Parameter Out Of Range (0xF4) will be returned.

Copyright Innovative Technology Ltd 2013

GET NOTE AMOUNT

There are 2 commands available to obtain the levels of the notes stored inside the SMART Payout.

Each denomination can individually be queried using Get Note Amount command (0x35). To get levels for all notes stored this needs to be sent as many time as there are denominations in the dataset (as detailed in the response of Setup Request).

Get Note Amount, 5.00 EUR
[35][F4 01 00 00][45 55 52]

OK, 3 Notes
[F0] [03 00]

Get Note Amount, 10.00
[35][E8 03 00 00][45 55 52]

OK, 2 Notes
[F0] [02 00]

The levels of all the notes in the dataset can be retrieved using the Get All Levels command (0x22). This command returns the levels of all denominations in the dataset programmed in the SMART Payout, along with the value and currency of the denomination.

This is typically more efficient than sending individual level requests.

Get All Levels [22]

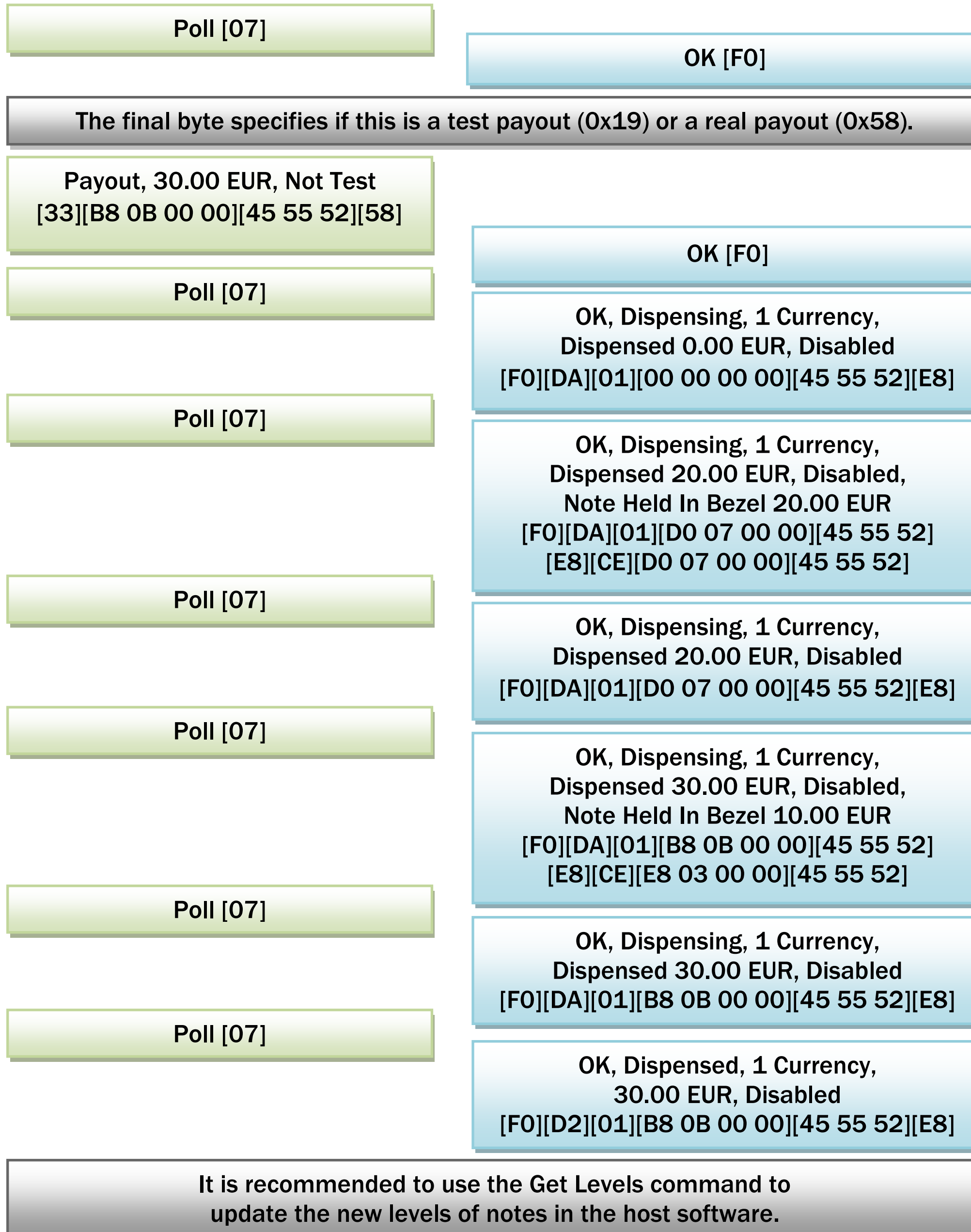
OK, 7 Denominations
3 x 5.00 EUR,
2 x 10.00 EUR,
5 x 20.00 EUR,
0 x 50.00 EUR,
0 x 100.00 EUR,
0 x 200.00 EUR,
0 x 500.00 EUR
[F0] [07]
[03 00] [F4 01 00 00] [45 55 52]
[02 00] [E8 03 00 00] [45 55 52]
[05 00] [D0 07 00 00] [45 55 52]
[00 00] [88 13 00 00] [45 55 52]
[00 00] [10 27 00 00] [45 55 52]
[00 00] [20 4E 00 00] [45 55 52]
[00 00] [50 C3 00 00] [45 55 52]

For both of the level commands detailed above, if the first byte of the response is not OK (0xF0) the possible error conditions are as follows.

- If the Payout is not detected as connected a Command Cannot Be Processed (0xF5) will be returned.
- If the payout is busy, Command Cannot Be Processed (0xF5) will be returned with an addition byte 0x03.

DISPENSING NOTES - PAYOUT

There are two methods to dispense notes. This example details Payout (command 0x33) which allows the Payout to decide which notes to payout based on options set within the device (command 0x50).



See error conditions section below for alternative responses.

DISPENSING NOTES - PAYOUT BY DENOMINATION

There are two methods to dispense notes. This example details Payout by Denomination (command 0x46) which allows the user to specify exactly which notes are paid out. This example pays 2 x €5, 1 x €20 notes.

The final byte specifies if this is a test payout (0x19) or a real payout (0x58).

Host	Device
Payout by denomination, 2 denominations 1 x 5.00 EUR, 1 x 20.00 EUR, Not Test [46] [02] [01 00][F4 01 00 00][45 55 52] [01 00][E8 03 00 00][45 55 52] [58]	OK [F0]
Poll [07]	OK, Dispensing, 1 Currency, Dispensed 0.00 EUR, Disabled [F0][DA][01][00 00 00 00][45 55 52][E8]
Poll [07]	OK, Dispensing, 1 Currency, Dispensed 20.00 EUR, Disabled, Note Held In Bezel 20.00 EUR [F0][DA][01][D0 07 00 00][45 55 52] [E8][CE][D0 07 00 00][45 55 52]
Poll [07]	OK, Dispensing, 1 Currency, Dispensed 20.00 EUR, Disabled [F0][DA][01][D0 07 00 00][45 55 52][E8]
Poll [07]	OK, Dispensing, 1 Currency, Dispensed 25.00 EUR, Disabled, Note Held In Bezel 5.00 EUR [F0][DA][01][C4 09 00 00][45 55 52] [E8][CE][E8 03 00 00][45 55 52]
Poll [07]	OK, Dispensing, 1 Currency, Dispensed 25.00 EUR, Disabled [F0][DA][01][C4 09 00 00][45 55 52][E8]
Poll [07]	OK, Dispensed, 1 Currency, 25.00 EUR, Disabled [F0][D2][01][C4 09 00 00][45 55 52][E8]

It is recommended to use the Get Levels command to update the new levels of notes in the host software.

Copyright Innovative Technology Ltd 2013

PAYOUT – ERROR CONDITIONS

The device will respond to Payout command (0x33) and Payout By Denomination command (0x46) with OK (0xF0) if there are no problems. Alternatively the following responses could be received:

- 0xF5 (command cannot be processed) is returned, an error code will follow. See the relevant command in section 9.3 for more details on these codes.
  - No additional byte
    - No Payout connected
  - 0x01
    - The validator has a dataset installed that does not match that of the notes stored in the recycler.
    - Payout error
  - 0x02 Can't pay exact value requested (request is higher than stored value or the value cannot be broken down with the notes available e.g. asking for €15 when no €5 notes are stored).
  - 0x03 Payout Busy
  - 0x04 Payout Disabled
- If the command was sent unencrypted then Parameter Out Of Range (0xF4) will be returned.
- If a non-valid currency code is given Parameter Out Of Range (0xF4) could be returned.

The test byte at the end of the payout command can be used in advance of actually issuing the payout command to check if the payout will be able to succeed.

FLOAT OPERATION - FLOAT

Floating routes notes to the cashbox/stacker and at end of the operation the SMART Payout should contain the number/value of notes specified. As with dispense, there are two methods to float notes. This example details Float (command 0x3D) which allows the Payout to decide which notes to float based on options set within the device (command 0x50). Most systems require the control afforded by Float By Denomination (0x44).

Minimum payout bytes specify there should be a way of paying this value at the end of the operation, all notes below this value can be routed to the cashbox.

The final byte specifies if this is a test float (0x19) or a real float (0x58).

Host	Device
Float, Minimum payout 0.10 EUR, Float Value 15.00 EUR, Not Test [3D][0A 00] [DC 05 00 00][45 55 52][58]	
	OK [F0]
Poll [07]	
	OK, Floating, 1 Currency, Routed 2.50 EUR [F0][D7][01][FA 00 00 00][45 55 52]
Poll [07]	
	OK, Floating, 1 Currency, Routed 9.75 EUR [F0][D7][01][CF 03 00 00][45 55 52]
Poll [07]	
	OK, Floating, 1 Currency, Routed 10.05 EUR [F0][D7][01][ED 03 00 00][45 55 52]
Poll [07]	
	OK, Floated, 1 Currency, Total Routed 10.05 EUR [F0][D8][01][ED 03 00 00][45 55 52]

It is recommended to use the Get Levels command to update the new levels of notes in the host software.

As with dispense commands, the response can potentially return a Command Unable To Be Processed (0xF5) response with an error byte detailing the cause of the error. See details in the payout section above.

FLOAT OPERATION – FLOAT BY DENOMINATION

Floating routes notes to the cashbox/stacker and at end of the operation the SMART Payout should contain the number/value of notes specified. As with dispense, there are two methods to float notes. This example details Float By Denomination (0x44) which enables the host to determine how many of each denomination of notes will be left in the Payout available for payout at the end of the operation. This is the recommended float operation for most applications.

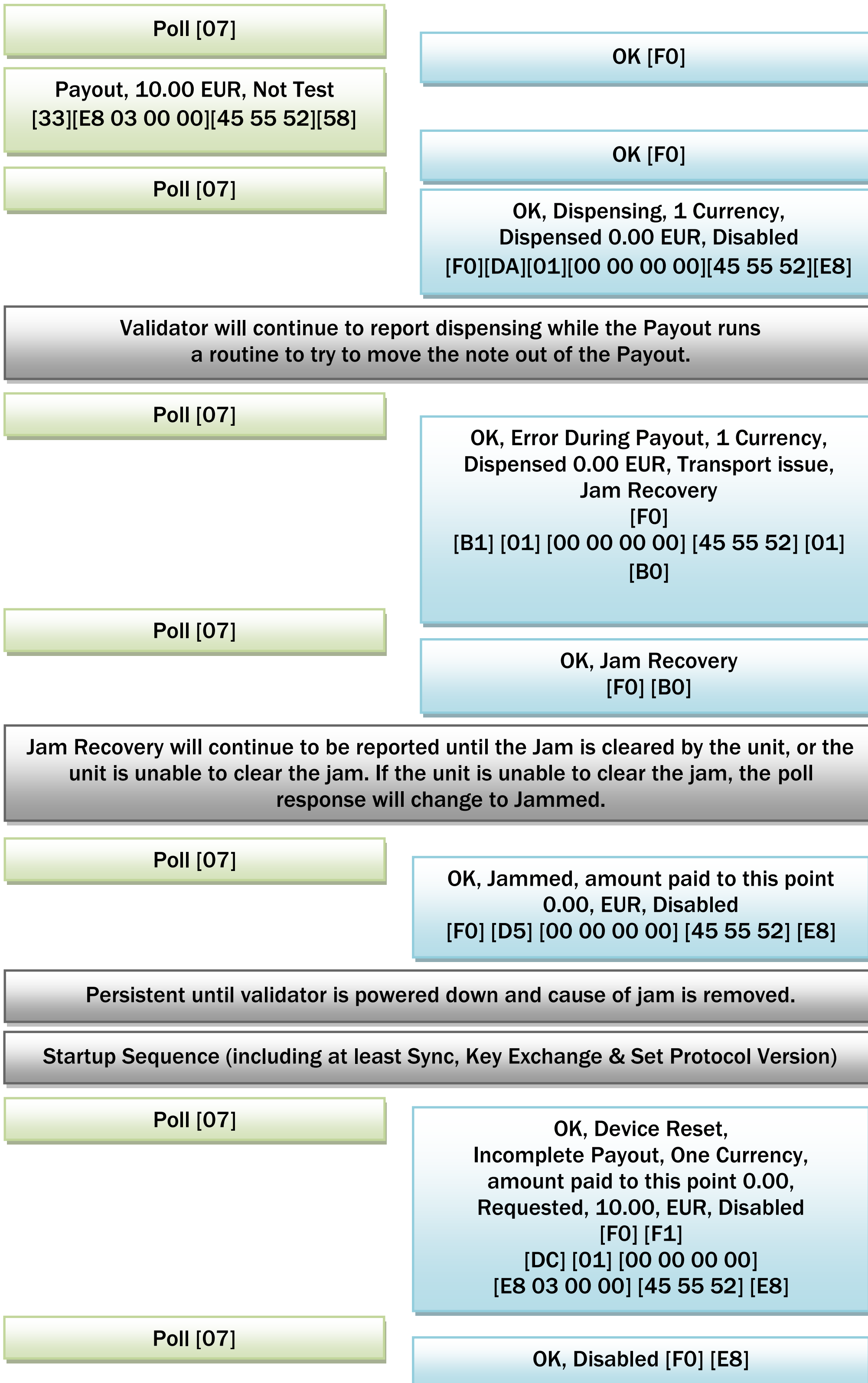
Minimum payout bytes specify there should be a way of paying this value at the end of the operation, all notes below this value can be routed to the cashbox.

The final byte specifies if this is a test float (0x19) or a real float (0x58).

Host	Device
Float By Denomination, 7 Coins, 0 x 0.02 EUR, 0 x 0.05 EUR, 0 x 0.10 EUR, 0 x 0.20 EUR, 75 x 0.50 EUR, 100 x 1.00 EUR, 50 x 2.00 EUR, Not Test [3D][07] [00 00][02 00 00 00][45 55 52] [00 00][05 00 00 00][45 55 52] [00 00][0A 00 00 00][45 55 52] [00 00][14 00 00 00][45 55 52] [4B 00][32 00 00 00][45 55 52] [64 00][64 00 00 00][45 55 52] [32 00][C8 00 00 00][45 55 52] [58]	
	OK [F0]
Poll [07]	
	OK, Floating, 1 Currency, Routed 16.57 EUR [F0][D7][01][79 60 00 00][45 55 52]
Poll [07]	
	OK, Floating, 1 Currency, Routed 29.85 EUR [F0][D7][01][A9 0B 00 00][45 55 52]
Poll [07]	
	OK, Floating, 1 Currency, Routed 37.22 EUR [F0][D7][01][8A 0E 00 00][45 55 52]
Poll [07]	
	OK, Floated, 1 Currency, Total Routed 37.22 EUR [F0][D8][01][8A 0E 00 00][45 55 52]

It is recommended to use the Get Levels command to update the new levels of notes in the host software.

As with dispense commands, the response can potentially return a Command Unable To Be Processed (0xF5) response with an error byte detailing the cause of the error. See details in the payout section above.

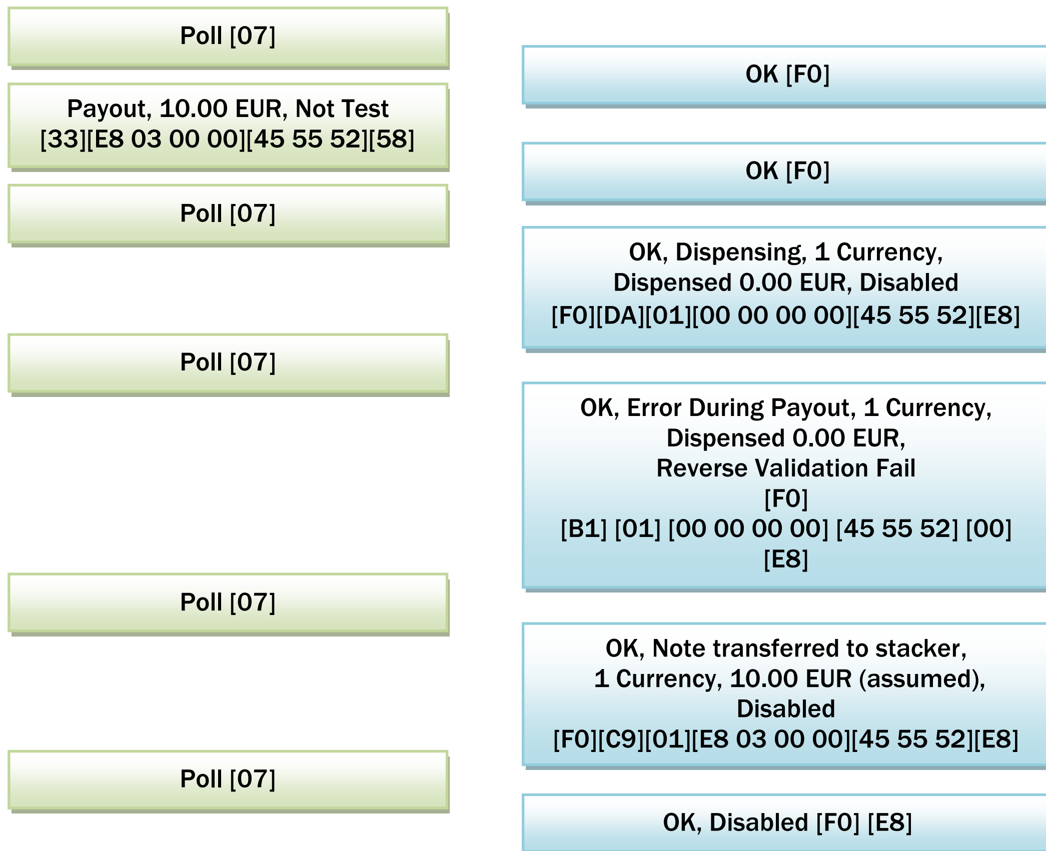


NOTE TRANSPORT ERROR DURING DISPENSE

REVERSE VALIDATION FAIL

Whilst moving notes out of the Payout storage (dispense, float or empty operations), the SMART Payout will use a procedure called “reverse validation” to read the note as it is moving down towards the cashbox to verify the value of the note that has retrieved from storage. If for some reason the note value is not correct it will fail the reverse validation check. In this case it will not be presented to the customer but instead stacked in the cashbox.

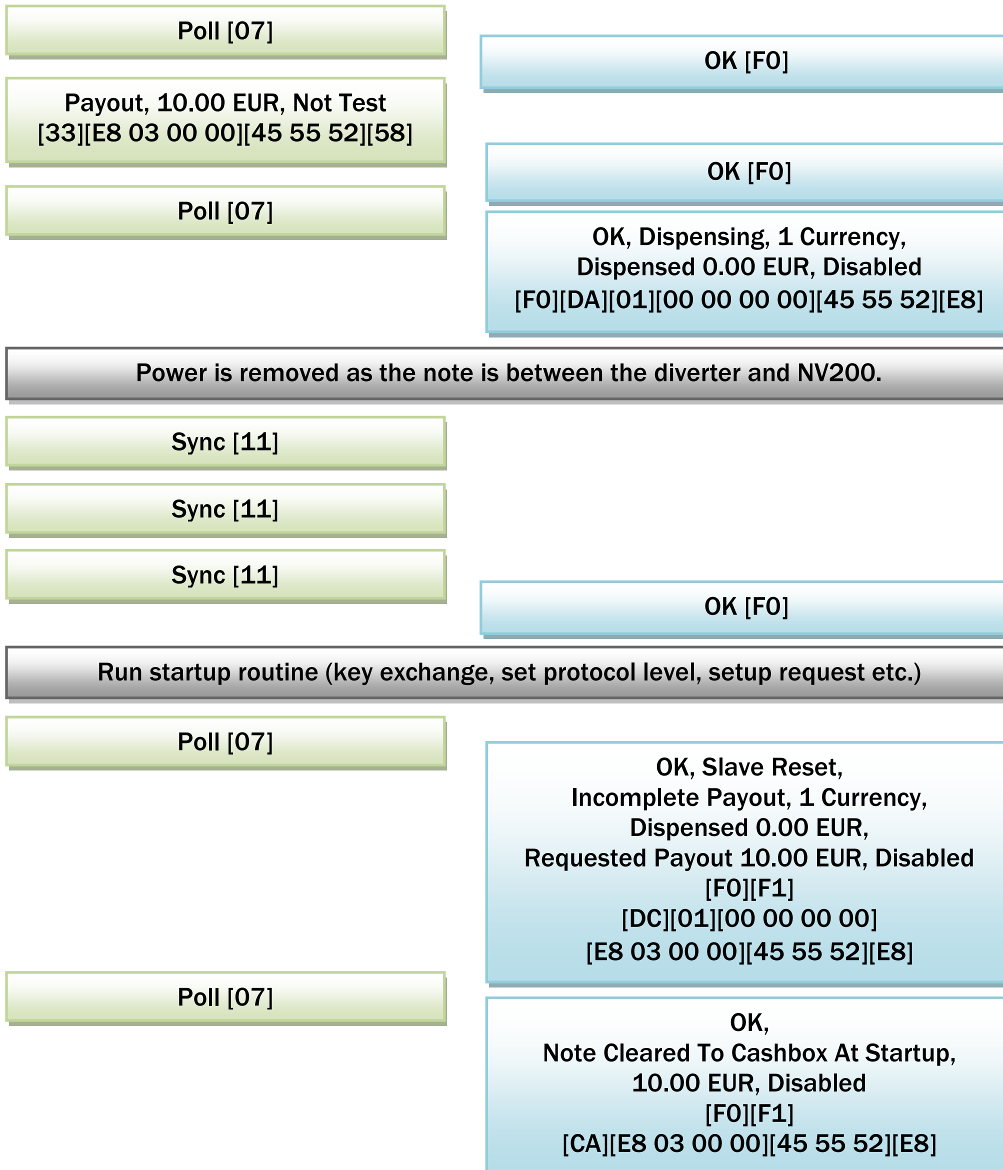
In the example below the SMART Payout is meant to dispense a €10 note however a €20 note is retrieved instead. The current operation will be halted and the device becomes disabled. The host can re-enable the device and re-try the payout if sufficient notes are left.



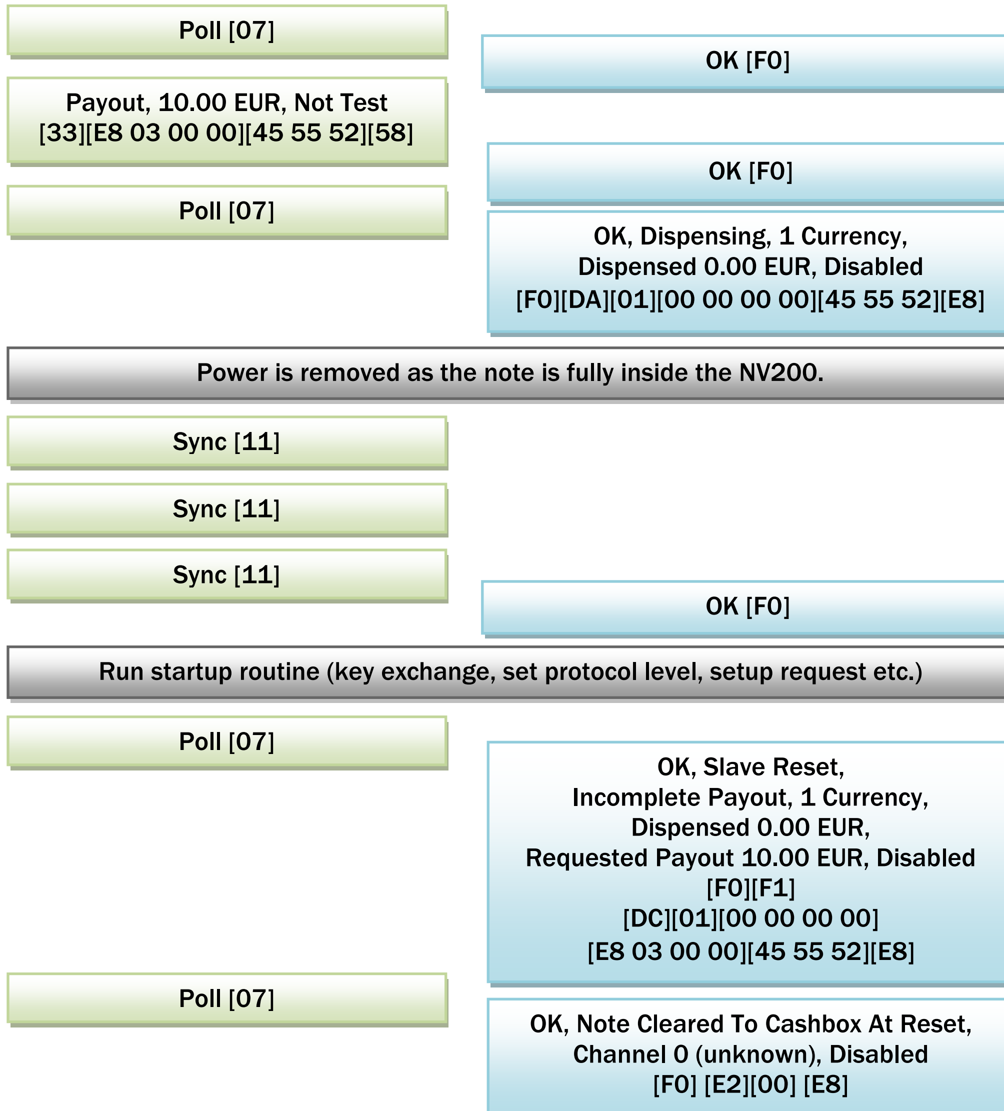
POWER REMOVED DURING PAYOUT

If the power is removed from the SMART Payout while it is dispensing the flow of events will depend on when the power is removed.

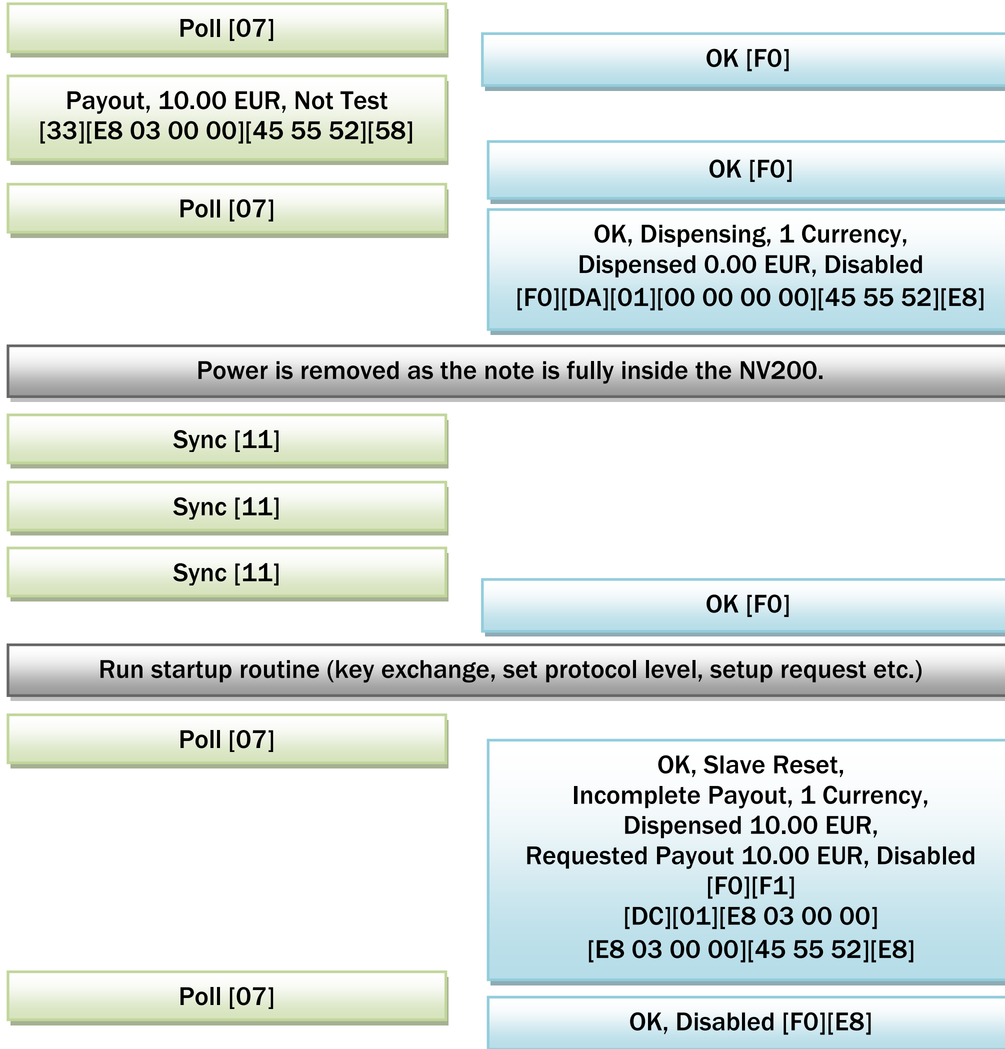
Here the flow is shown if the power is removed before the note is fully inside the NV200 and is still partly in the Payout.



If the power is removed once the note is inside the NV200, but not yet reverse validated and being returned to the customer; the flow will be as follows.



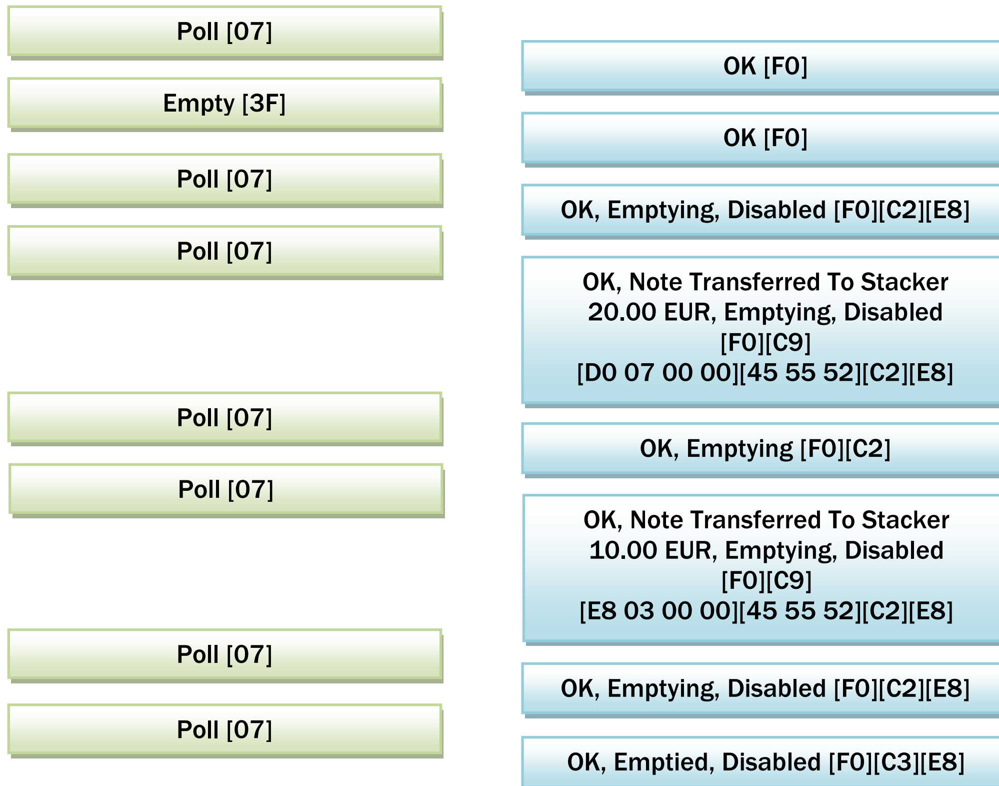
If the power is removed once the note is inside the NV200, validated and in the process of being returned to the customer; the flow will be as follows.



Copyright Innovative Technology Ltd 2013

EMPTY PAYOUT – EMPTY

There are two options when emptying a SMART Payout, the first option is the Empty (0x3F) command. This does not keep track of the notes as they are moved to the cashbox/stacker.

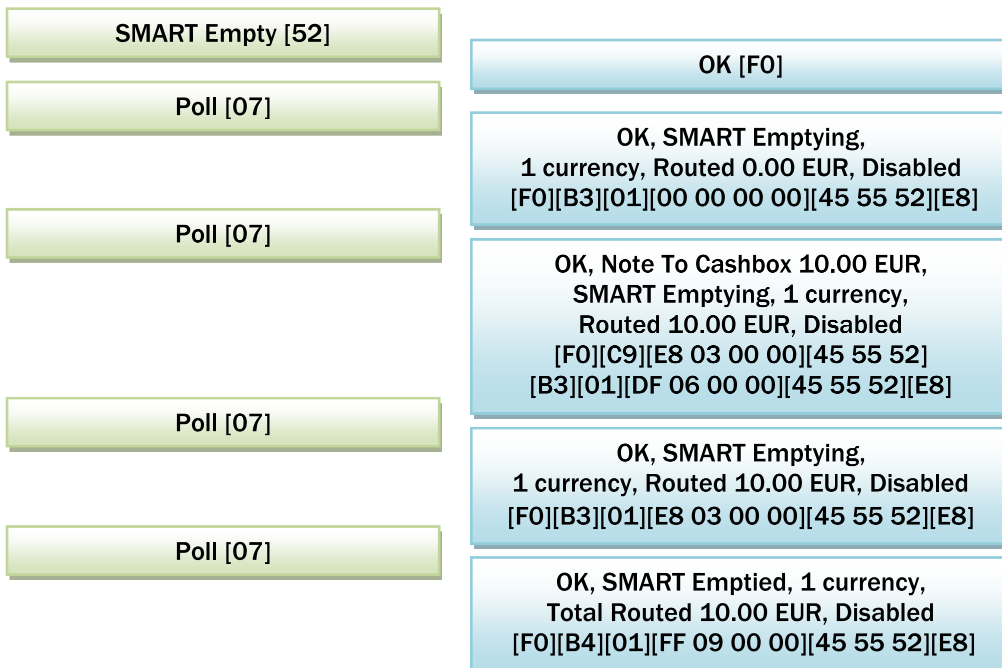


An OK (0xF0) response to the empty or SMART Empty commands indicates the empty operation was accepted and will begin. The following error codes could also be returned.

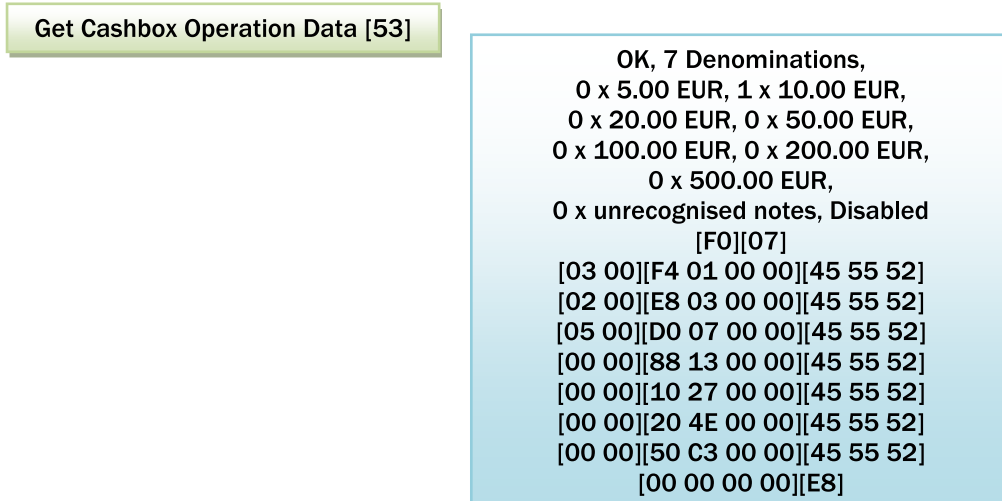
- 0xF5 (command cannot be processed) is returned, an error code will follow. See the relevant command in section 9.3 for more details on these codes.
    - No additional byte - No Payout connected
    - 0x03 - Payout Busy
- If the command was sent unencrypted then Parameter Out Of Range (0xF4) will be returned.

EMPTY PAYOUT – SMART EMPTY

If the host requires a summary of the notes that were moved to the cashbox during the empty operation then the SMART Empty command (0x52) can be used. During this procedure, all notes in the Recycler storage are moved to the cashbox/stacker. At the end of the procedure the Get Cashbox Operation Data command (0x53) will report a summary of the notes moved to the cashbox.



The Get Cashbox Operation Data (command 0x53) can now be sent to determine which coins were deposited into the cashbox during the operation.



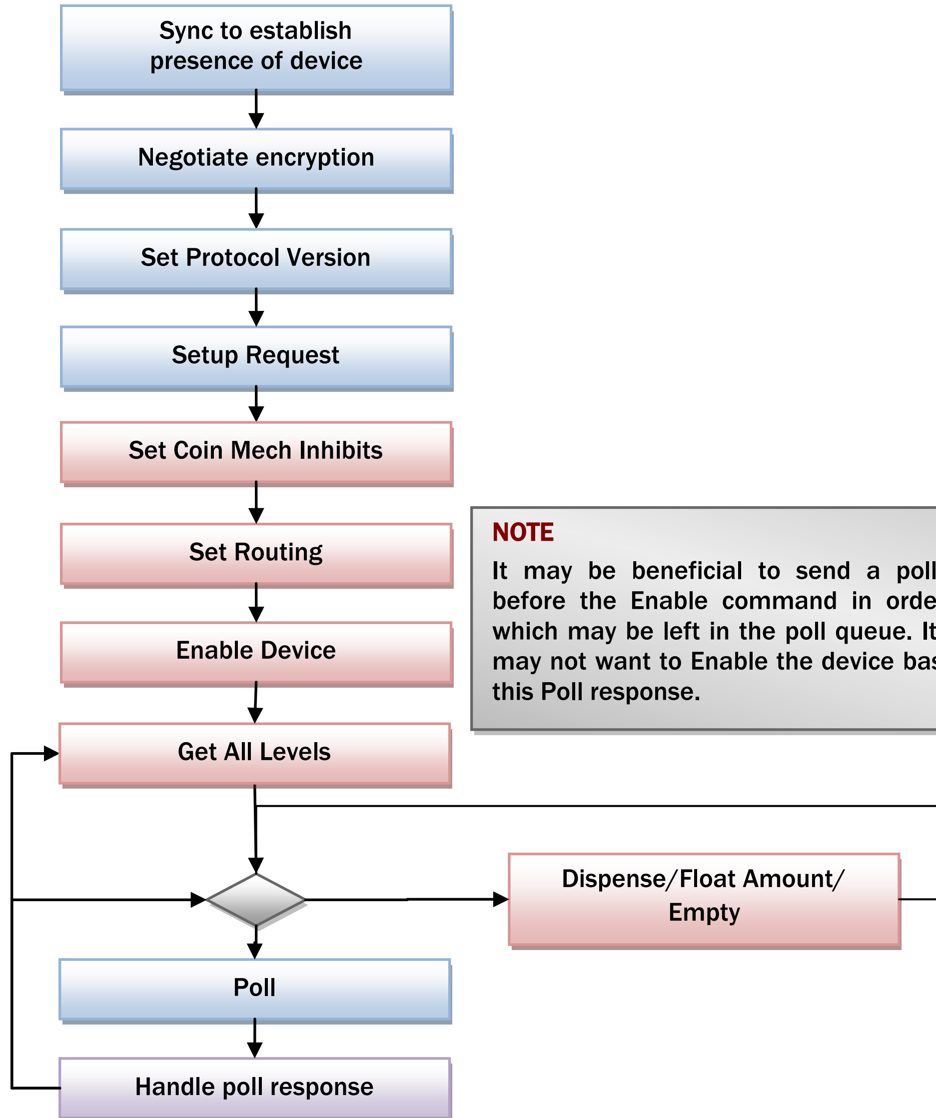
Note: Get Cashbox Operation Data can be set on completion of any Float, Dispense or SMART Empty operations.

Error conditions are detailed above.

8.6 SMART HOPPER

OVERVIEW OF OPERATION

This flow outlines the blocks that make up the fundamental operation of a SMART Hopper.



NOTE

It may be beneficial to send a poll command immediately before the Enable command in order to process any events which may be left in the poll queue. It is possible that the host may not want to Enable the device based on what it receives in this Poll response.

SETUP REQUEST

Host Protocol Version [06] [08]

OK [F0]

Setup Request [05]

OK, Setup Data [F0]
[F0] [03 30 36 31 34
45 55 52 07 07
02 00 05 00 0A 00
14 00 32 00 64 00
C8 00
45 55 52 45 55 52 45 55 52
45 55 52 45 55 52 45 55 52
45 55 52]

Parse setup request

03 = Unit Type (SMART Hopper)
30 36 31 34 = Firmware (6.14)
45 55 52 = Country Code (EUR)
07 = Protocol Version (7)
07 = Number of channels (7)
02 00 00 00 = Value of Ch1 (0.02)
05 00 00 00 = Value of Ch2 (0.05)
0A 00 00 00 = Value of Ch3 (0.10)14 00
00 00 = Value of Ch4 (0.20)
32 00 00 00 = Value of Ch5 (0.50)
64 00 00 00 = Value of Ch6 (1.00)
C8 00 00 00 = Value of Ch7 (2.00)
45 55 52 (repeated x 4) = Currency Code
for each channel (EUR)

SETTING COIN MECHANISM INHIBITS

By default, at startup, all coins available to the coin mechanism that are supported by the SMART Hopper dataset are enabled and the Master Inhibit is disabled so coins will be accepted.

To disable all coins, enable the global coin mechanism inhibit (command 0x49). To disable specific coins, use specific coin inhibits (command 0x40).

Individual coins can be inhibited using Set Coin Mech Inhibits command. The byte following the command should be 0 to disable the coin so none are accepted, 1 will enable the coin.

Set Coin Mech Inhibit,
Enable Coin, 0.20 EUR
[40][01][14 00 00 00][45 55 52]

OK [F0]

If OK (0xF0) is returned, the command was successful and the inhibits will be set. If no coin mechanism is detected Wrong Number of Parameters (0xF3) will be returned.

Alternatively all coins can be inhibited using Set Coin Mech Global Inhibit command. The byte following the command should be 0 to disable the mechanism so no coins are accepted, 1 will enable the mechanism.

Set Coin Mech Global Inhibit,
Enable Acceptance
[49][01]

OK [F0]

If no coin mechanism is detected Wrong Number of Parameters (0xF3) will be returned.

ROUTING COINS

Routing changes the path for the coins as they are read. The available routes are:

- Recycle into the Hopper storage and use for pay-outs.
- Send to the cashbox to remove from Hopper storage.

Routing codes:
0x00 – recycle & use for payout
0x01 – cashbox

Set Routing, Recycle, 0.50 EUR
[3B][00][32 00 00 00][45 55 52]

OK [F0]

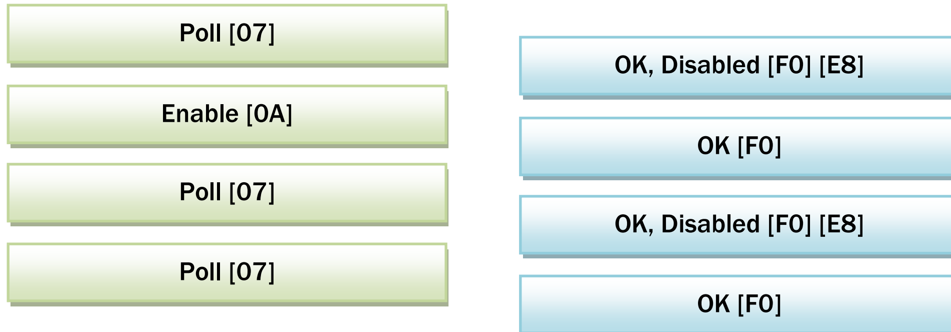
Set Routing, To cashbox, 0.05 EUR
[3B][01][05 00 00 00][45 55 52]

OK [F0]

- OK (0xF0) indicates setting the route was successful.
- Parameter Out Of Range (0xF4) is reported if the command is sent unencrypted.
- Command Cannot be Processed is reported if there was a problem processing the command.

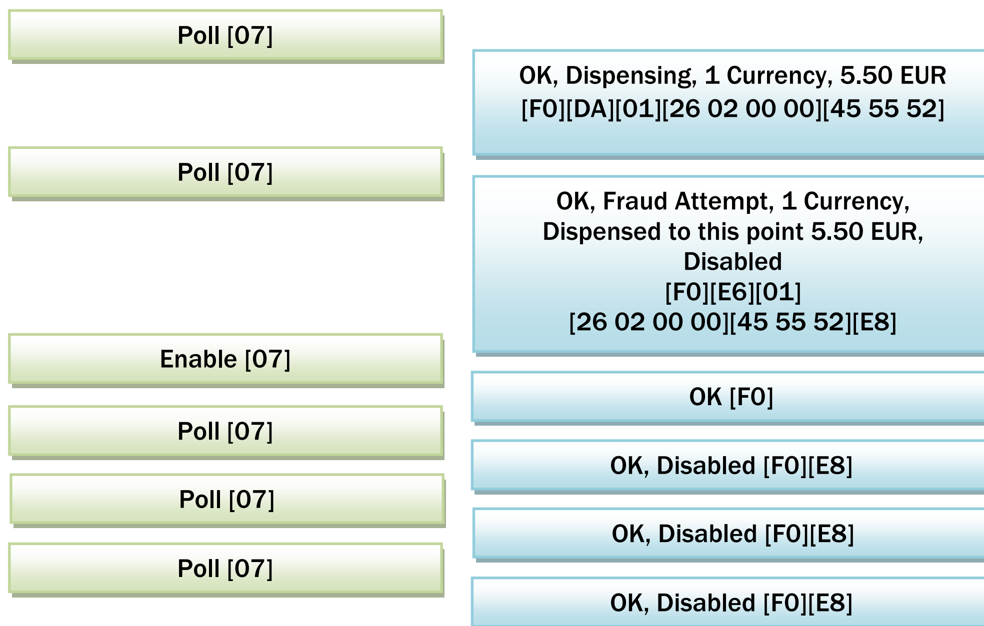
ENABLE HOPPER

Enables the SMART Hopper.



Enable will always respond ok, use Poll command (0x07) to check the SMART Hopper has enabled. The Disabled event is reported for one Poll response after the Enable command is received. This is so that the host is informed the device has been disabled.

The SMART Hopper will not enable if it is currently reporting Fraud Attempt Seen (0xD5).



GET COIN AMOUNT

There are 2 commands available to obtain the levels of the coins stored inside the SMART Hopper.

Each denomination can individually be queried using Get Coin Amount command (0x35). To get levels for all coins stored this needs to be sent as many time as there are denomination in the dataset (as detailed in Setup Request).

Get Coin Amount, 0.50 EUR
[35][32 00 00 00][45 55 52]

OK, 30 coins
[F0] [1E 00]

Get Coin Amount, 1.00
[35][64 00 00 00][45 55 52]

OK, 520 coins
[F0] [08 02]

The levels of all the coins in the dataset can be retrieved using the Get All Levels command (0x22). This command returns the levels of all denominations in the dataset programmed in the SMART Hopper, along with the value and currency of the denomination.

This is typically more efficient than sending individual level requests.

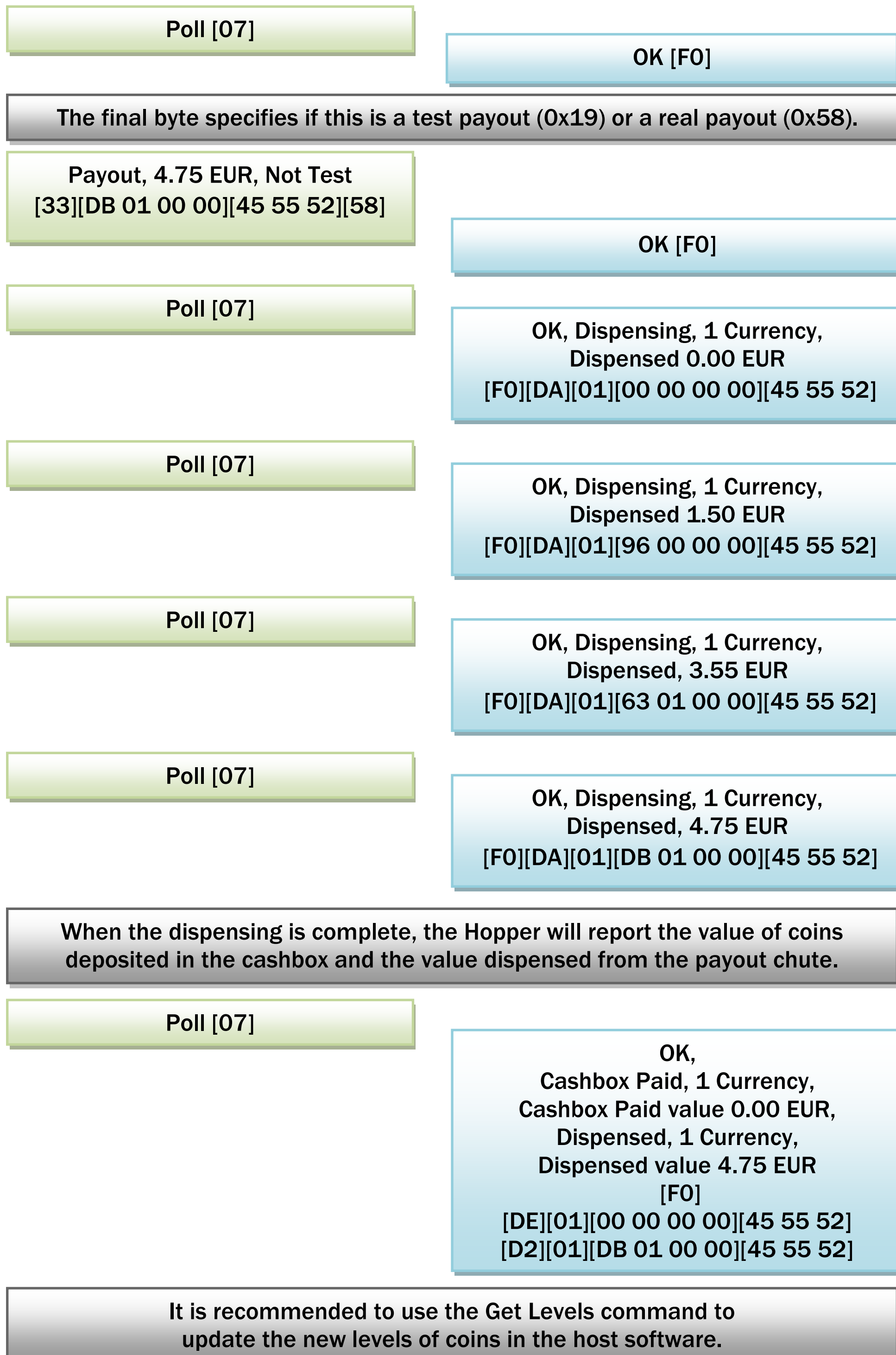
Get All Levels [22]

OK, 7 Denominations
0 x 0.02 EUR,
0 x 0.05 EUR,
0 x 0.10 EUR,
0 x 0.20 EUR,
30 x 0.50 EUR,
520 x 1.00 EUR,
00 x 2.00 EUR
[F0] [07]
[00 00][02 00 00 00][45 55 52]
[00 00][05 00 00 00][45 55 52]
[00 00][0A 00 00 00][45 55 52]
[00 00][14 00 00 00][45 55 52]
[1E 00][32 00 00 00][45 55 52]
[08 02][64 00 00 00][45 55 52]
[00 00][C8 00 00 00][45 55 52]

For both of the level commands detailed above, if the payout unable to process the command, Command Cannot Be Processed (0xF5) will be returned.

DISPENSING COINS - PAYOUT

There are two methods to dispense coins. This example details Payout (command 0x33) which allows the Hopper to decide which coins to payout based on options set within the device (command 0x50).



DISPENSING COINS - PAYOUT BY DENOMINATION

There are two methods to dispense coins. This example details Payout by Denomination (command 0x46) which allows the user to specify exactly which coins are paid out. This example pays 4 x €1.00, 1 x €0.50, 1 x €0.20 and 1 x €0.05 coins.

The final byte specifies if this is a test payout (0x19) or a real payout (0x58).

Host	Device
Payout by denomination, 4 denominations 1 x 0.05 EUR, 1 x 0.20 EUR 1 x 0.50 EUR, 4 x 1.00 EUR, Not Test [46] [04] [01 00][05 00 00 00][45 55 52] [01 00][14 00 00 00][45 55 52] [01 00][32 00 00 00][45 55 52] [04 00][64 00 00 00][45 55 52] [58]	
	OK [F0]
Poll [07]	
	OK, Dispensing, 1 Currency, Dispensed 0.00 EUR [F0][DA][01][00 00 00 00][45 55 52]
Poll [07]	
	OK, Dispensing, 1 Currency, Dispensed 1.05 EUR [F0][DA][01][69 00 00 00][45 55 52]
Poll [07]	
	OK, Dispensing, 1 Currency, Dispensed 3.75 EUR [F0][DA][01][77 01 00 00][45 55 52]
Poll [07]	
	OK, Dispensing, 1 Currency, Dispensed 4.75 EUR [F0][DA][01][DB 01 00 00][45 55 52]

When the dispensing is complete, the Hopper will report the value of coins deposited in the cashbox and the value dispensed from the payout chute.

Host	Device
Poll [07]	
	OK, Cashbox Paid, 1 Currency, Cashbox Paid value 0.00 EUR, Dispensed, 1 Currency, Dispensed value 4.75 EUR [F0] [DE][01][00 00 00 00][45 55 52] [D2][01][DB 01 00 00][45 55 52]

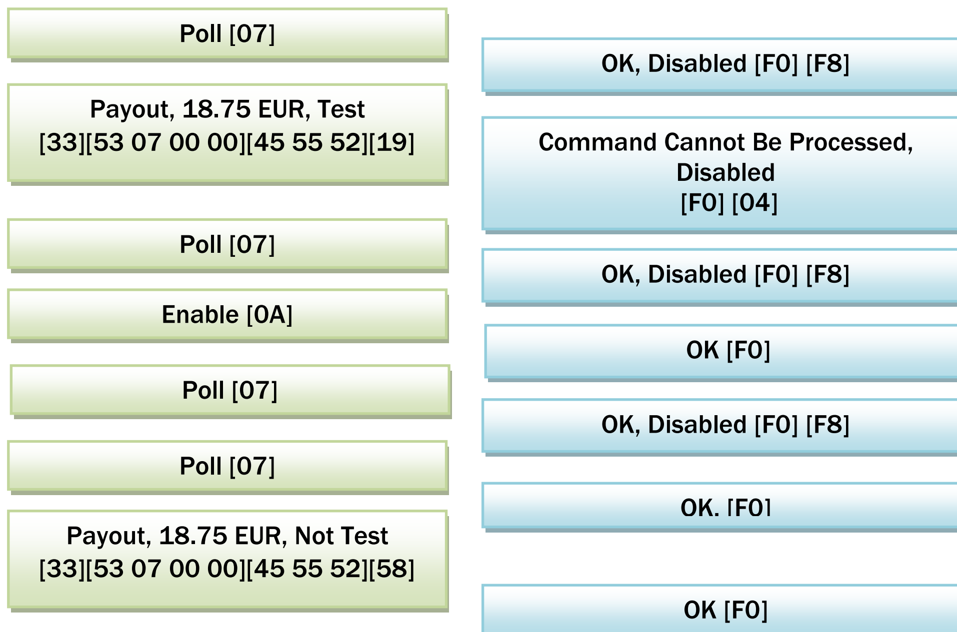
It is recommended to use the Get Levels command to update the new levels of coins in the host software.

PAYOUT – ERROR CONDITIONS

The device will respond to Payout command (0x33) and Payout By Denomination command (0x46) with OK (0xF0) if there are no problems. Alternatively the following responses could be received:

- 0xF5 (command cannot be processed) is returned, an error code will follow. See the relevant command in section 9.3 for more details on these codes.
  - 0x01 Not enough value in the SMART Hopper to complete the payout.
  - 0x02 Can't pay exact value requested (request is higher than stored value or the value cannot be broken down with the coins available e.g. asking for €0.15 when no €0.05, €0.02, or €0.01 coins are stored).
  - 0x03 SMART Hopper Busy
  - 0x04 SMART Hopper Disabled
- If the command was sent unencrypted then Parameter Out Of Range (0xF4) will be returned.

The test byte at the end of the payout command can be used in advance of actually issuing the payout command to check if the payout will be able to succeed.



FLOAT OPERATION - FLOAT

Floating routes coins to the cashbox below and at end of the operation the SMART Hopper should contain the number/value of coins specified. As with dispense, there are two methods to float coins. This example details Float (command 0x3D) which allows the Hopper to decide which coins to float based on options set within the device (command 0x50). Most systems require the control afforded by Float By Denomination (0x44).

Minimum payout bytes specify there should be a way of paying this value at the end of the operation, all coins below this value can be routed to the cashbox.

The final byte specifies if this is a test float (0x19) or a real float (0x58).

Host	Device
Float, Minimum payout 0.10 EUR, Float Value 15.00 EUR, Not Test [3D][0A 00] [DC 05 00 00][45 55 52][58]	
	OK [F0]
Poll [07]	
	OK, Floating, 1 Currency, Routed 2.50 EUR [F0][D7][01][FA 00 00 00][45 55 52]
Poll [07]	
	OK, Floating, 1 Currency, Routed 9.75 EUR [F0][D7][01][CF 03 00 00][45 55 52]
Poll [07]	
	OK, Floating, 1 Currency, Routed 10.05 EUR [F0][D7][01][ED 03 00 00][45 55 52]
Poll [07]	
	OK, Floated, 1 Currency, Total Routed 10.05 EUR [F0][D8][01][ED 03 00 00][45 55 52]

It is recommended to use the Get Levels command to update the new levels of coins in the host software.

As with dispense commands, the response can potentially return a Command Unable To Be Processed (0xF5) response with an error byte detailing the cause of the error. See details in the payout section above.

FLOAT OPERATION – FLOAT BY DENOMINATION

Floating routes coins to the cashbox below and at end of the operation the SMART Hopper should contain the number/value of coins specified. As with dispense, there are two methods to float coins. This example details Float By Denomination (0x44) which enables the host to determine how many of each denomination of coin will be left in the Hopper available for payout at the end of the operation. This is the recommended float operation for most applications.

Minimum payout bytes specify there should be a way of paying this value at the end of the operation, all coins below this value can be routed to the cashbox.

The final byte specifies if this is a test float (0x19) or a real float (0x58).

Float By Denomination, 7 Coins,
0 x 0.02 EUR, 0 x 0.05 EUR,
0 x 0.10 EUR, 0 x 0.20 EUR,
75 x 0.50 EUR, 100 x 1.00 EUR,
50 x 2.00 EUR, Not Test
[3D][07]
[00 00][02 00 00 00][45 55 52]
[00 00][05 00 00 00][45 55 52]
[00 00][0A 00 00 00][45 55 52]
[00 00][14 00 00 00][45 55 52]
[4B 00][32 00 00 00][45 55 52]
[64 00][64 00 00 00][45 55 52]
[32 00][C8 00 00 00][45 55 52]
[58]

OK [F0]

Poll [07]

OK, Floating, 1 Currency,
Routed 16.57 EUR
[F0][D7][01][79 60 00 00][45 55 52]

Poll [07]

OK, Floating, 1 Currency,
Routed 29.85 EUR
[F0][D7][01][A9 0B 00 00][45 55 52]

Poll [07]

OK, Floating, 1 Currency,
Routed 37.22 EUR
[F0][D7][01][8A 0E 00 00][45 55 52]

Poll [07]

OK, Floated, 1 Currency,
Total Routed 37.22 EUR
[F0][D8][01][8A 0E 00 00][45 55 52]

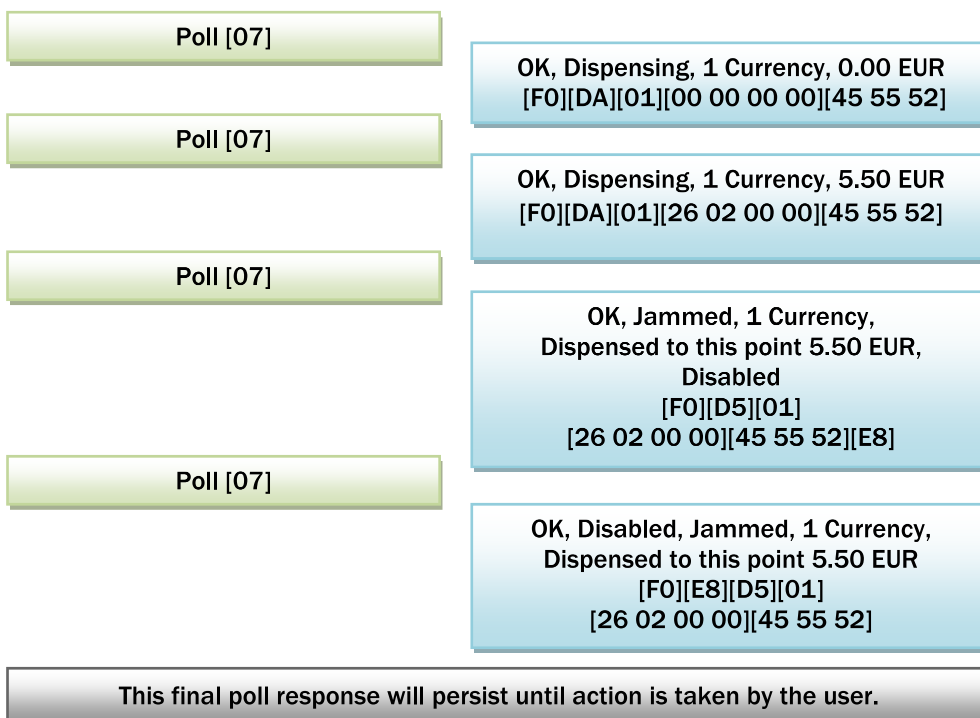
It is recommended to use the Get Levels command to update the new levels of coins in the host software.

As with dispense commands, the response can potentially return a Command Unable To Be Processed (0xF5) response with an error byte detailing the cause of the error. See details in the payout section above.

COIN JAM

In the event that a coin is in the position that it blocks the normal movement of the motors, the SMART Hopper will start a routine that will attempt to recover normal movement. During this time, Dispensing (0xDA), Floating (0xD7) or Empting (0xC2) will continue to be returned as a poll response. In the rare event that this routine cannot move the coin that is blocking the movement the jammed (0xD5) poll response will be reported with the amount that was dispensed at the point the jam occurred.

Jams can be recovered by removing power, manually tipping all coins out of the SMART Hopper and clearing the jam. When the power is re-applied the SMART Hopper will initialise and, if the jam is clear return to normal operation. Once keys are negotiated and the SMART Hopper is enabled the host can send a command to resume the operation.



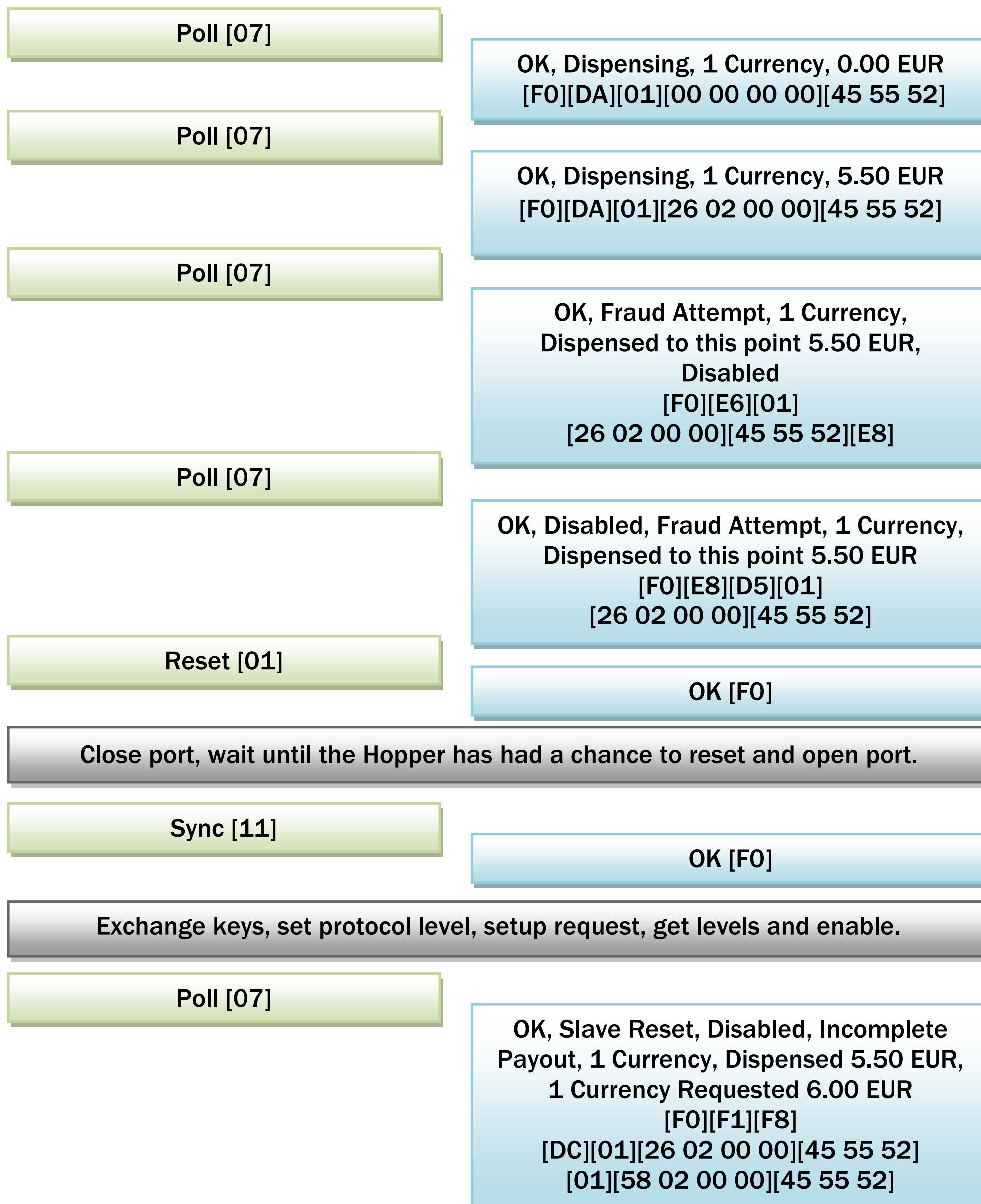
This final poll response will persist until action is taken by the user.

FRAUD ATTEMPT

If a sensor is triggered out of order it could be an indication that a manipulation is being performed to the Hopper. In this case a Fraud Attempt poll event is reported and, if in progress, payout/float/empty routine is halted.

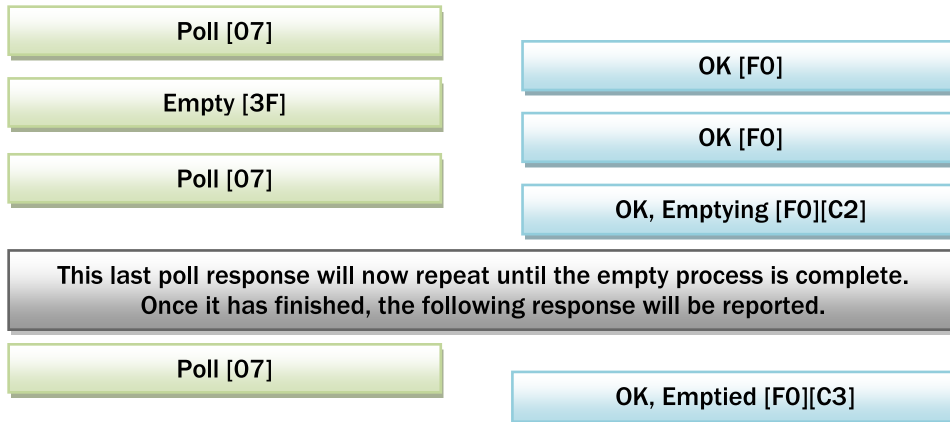
The Fraud Event response is persistent and can only be cleared by resetting the device. At this time the routine can be continued.

If multiple fraud attempts are reported in a short time it is advised that the host disables payout for security reasons until the cause can be investigated.



EMPTY HOPPER - EMPTY

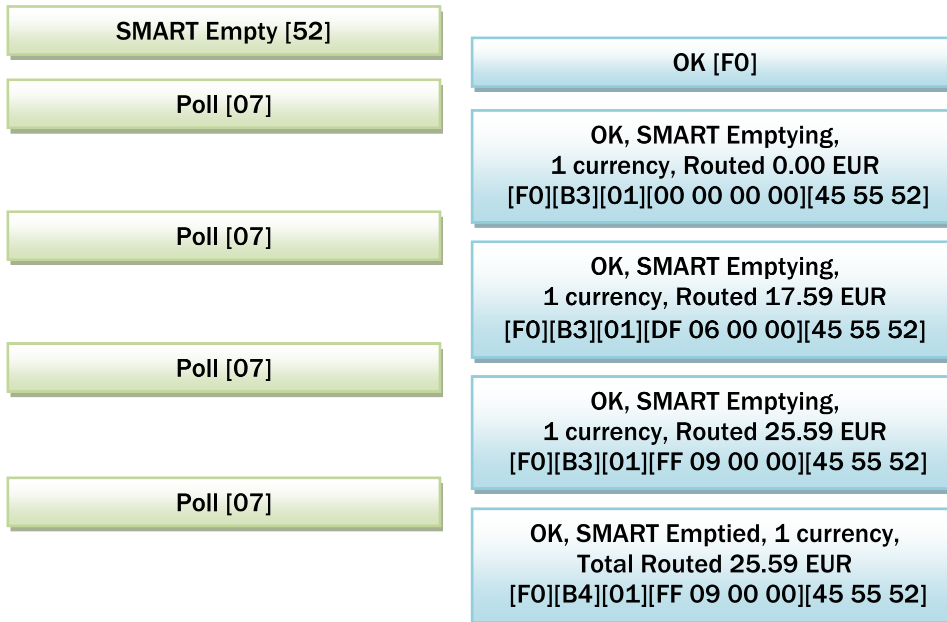
There are two options when emptying a SMART Hopper, the first option is the Empty (0x3F) command. This does not keep track of the coins as they are moved to the cashbox.



The device will respond to Empty command (0x3F) and SMART Empty command (0x52) with OK (0xF0) if there are no problems. Alternatively if the command was sent unencrypted then Parameter Out Of Range (0xF4) will be returned.

EMPTY HOPPER – SMART EMPTY

If the host requires a summary of the coins that were moved to the cashbox during the empty operation then the SMART Empty command (0x52) can be used. During this procedure, all coins in the Hopper storage are moved to the cashbox, as with the empty routine however they are counted out. At the end of the procedure the Get Cashbox Operation Data command (0x53) will report a summary of the coins moved to the cashbox.



The Get Cashbox Operation Data (command 0x53) can now be sent to determine which coins were deposited into the cashbox during the operation.

Get Cashbox Operation Data [53]

OK, 7 Denominations,
2 x 0.02 EUR, 1 x 0.05 EUR,
0 x 0.10 EUR, 0 x 0.20 EUR,
5 x 0.50 EUR, 3 x 1.00 EUR,
10 x 2.00 EUR,
1 x unrecognised coins
[F0][07]
[02 00][02 00 00 00][45 55 52]
[01 00][05 00 00 00][45 55 52]
[00 00][0A 00 00 00][45 55 52]
[00 00][14 00 00 00][45 55 52]
[05 00][32 00 00 00][45 55 52]
[03 00][64 00 00 00][45 55 52]
[0A 00]C8 00 00 00][45 55 52]
[01 00 00 00]

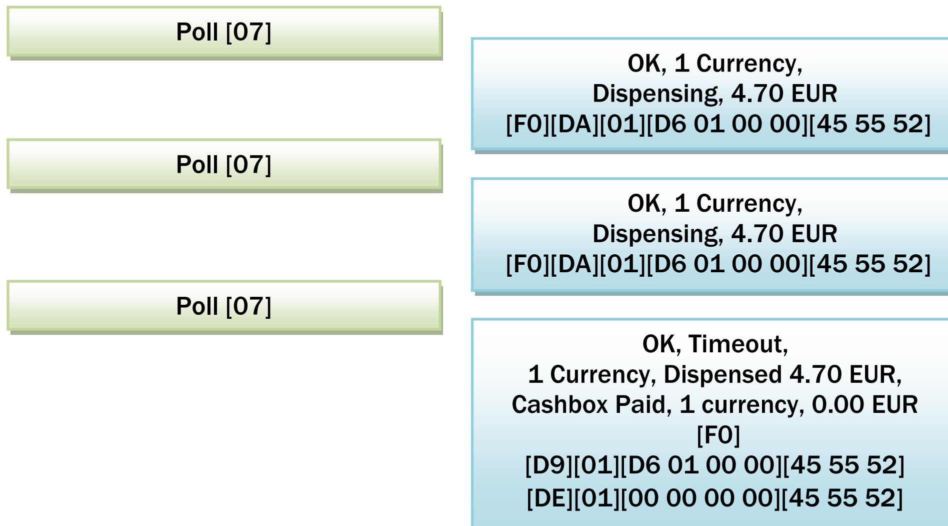
Note: Get Cashbox Operation Data can be set on completion of any Float, Dispense or SMART Empty operations.

Copyright Innovative Technology Ltd 2013

TIMOUT DURING PAYOUT

If the SMART Hopper is searching for a specific coin as part of a payout or float operation it has a timeout set. Once this timeout is reached and the coin has not been found the Hopper will return the poll response Timeout (0xD9).

The operation can be retried with a different selection of coins or handled in another way, specific to the host device/application.



COINS INSERTED TO COIN MECHANISM

Poll [07]

OK [F0]

Poll [07]

OK, Coin Credit, 2.00 EUR
[F0][DF][C8 00 00 00][45 55 52]

It is recommended to update the new levels of coins in the host software.

9 COMMANDS FOR ITL DEVICES

9.1 BANK NOTE VALIDATOR (NV9USB, NV10USB, BV20, BV50, BV100, NV200)

RESET (0X01)

Single byte command causes the unit to reset.

HOST PROTOCOL VERSION (0X06)

Two byte command sets the unit to report events up to and including those found in the specified protocol version. Please note that the highest protocol version that a unit will support is determined by its firmware. Please see the appendix for more information.

POLL (0X07)

Single byte command instructs the unit to report all the events that have occurred since the last time a poll was sent to the unit. For a more detailed explanation of the poll command and polling the unit, please see section 7 - Polling Devices.

GET SERIAL NUMBER (0X0C)

Single byte command causes the unit to report its unique serial number.
Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Serial number. 4 bytes. Big endian.	0x00	This serial number as a 32 bit integer is 2746699.
2		0x29	
3		0xE9	
4		0x4B	

SYNCHRONISATION COMMAND (0X11)

This single byte command tells the unit that the next sequence ID will be 1. This is always the first command sent to a unit, to prepare it to receive any further commands.

DISABLE (0X09)

This single byte command disables the unit. This means the unit will enter its disabled state and not execute any further commands or perform any other actions. A poll to the unit while in this state will report disabled (0xE8).

ENABLE (0X0A)

Single byte command enables the unit. It will now respond to and execute commands.

DISPLAY ON (0X03)

Single byte command turns on the bezel light when the unit is enabled.

DISPLAY OFF (0X04)

Single byte command turns off the bezel light when the unit is enabled.

REJECT (0X08)

Single byte command causes the validator to reject the current note.

SETUP REQUEST (0X05)

For general information about the setup request see section 6.3.2. Single byte command. The below table displays the response data of the setup request and provides an example of a real response from a unit with a Euro note dataset.

Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Unit type. Single byte.	0x00	Note validator type.
2	Firmware version. 4 bytes. Each byte represents an ASCII character.	0x00	This gives 0333 when converted to ASCII characters. Formatted it would read 3.33.
3		0x33	
4		0x33	
5		0x33	
6	Country code of validator. 3 bytes. Each byte represents an ASCII character.	0x45	When converted to ASCII characters it reads EUR.
7		0x55	
8		0x52	
9	Value multiplier. 3 bytes. Big endian.	0x00	When converted into a 24 bit integer it will have the value 1.
10		0x00	
11		0x01	
12	Number of channels.	0x04	Four channels.
13*	Channel values. Single byte per channel.	0x05	Five.
14		0x0A	Ten.
15		0x14	Twenty.
16		0x32	Fifty.
17*	Channel security level. Single byte per channel. (Legacy code, now deprecated).	0x02	Level 2.
18		0x02	Level 2.
19		0x02	Level 2.
20		0x02	Level 2.
21*	Real value multiplier. 3 bytes. Big endian.	0x00	When converted into a 24 bit integer it will have the value 100.
22		0x00	
23		0x64	
24*	Protocol version.	0x07	Protocol version 7.
25*	Country codes for each channel. 3 bytes per channel.	0x45	Each channel has the country code EUR when converted to an ASCII character.
26		0x55	
27		0x52	
28		0x45	
29		0x55	
30		0x52	
31		0x45	
32		0x55	
33		0x52	
34		0x45	
35		0x55	
36		0x52	
37*	Channel values for each channel. 4 bytes per channel. Little endian.	0x05	When converted to a 32 bit integer, these values come out at 5, 10, 20, 50.
38		0x00	
39		0x00	
40		0x00	
41		0x0A	
42		0x00	
43		0x00	
44		0x00	

Copyright Innovative Technology Ltd 2013

45		0x14	
46		0x00	
47		0x00	
48		0x00	
49		0x32	
50		0x00	
51		0x00	
52		0x00	

* - These sections' start position and length in the array will vary depending on the number of channels.

UNIT DATA (0X0D)

Single byte command causes the validator to return information about itself. It is similar to the Setup Request command but a more concise version. It is intended for host machines with limited resources. The below table displays the response data of the unit data request and provides an example of a real response from a unit with a Euro note dataset.
Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Unit type. Single byte.	0x00	Note validator type.
2	Firmware version. 4 bytes. Each byte represents an ASCII character.	0x00	This gives 0333 when converted to ASCII characters. Formatted it would read 3.33.
3		0x33	
4		0x33	
5		0x33	
6	Country code of validator. 3 bytes. Each byte represents an ASCII character.	0x45	When converted to ASCII characters it reads EUR.
7		0x55	
8		0x52	
9	Value multiplier. 3 bytes. Big endian.	0x00	When converted into a 24 bit integer it will have the value 1.
10		0x00	
11		0x01	
12	Protocol version.	0x07	Protocol version 7.

CHANNEL VALUE DATA (0X0E)

Single byte command causes the validator to return the number of channels it is using followed by the value of each channel. The below table displays the response data of the channel value data request and provides an example of a real response from a unit with a Euro note dataset.
Return data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Number of channels. Single byte.	0x04	Four channels.
2*	Channel values. Single byte per channel.	0x05	Five.
3		0x0A	Ten.
4		0x14	Twenty.
5		0x32	Fifty.

* - This section's length in the array will vary depending on the number of channels.

LAST REJECT CODE (0X17)

Single byte command causes the validator to report the reason for the last note being rejected.
Return data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Reject code. Single byte.	0x01	Note length incorrect. For a full list of reject codes see **Appendix**

HOLD (0X18)

Single byte command causes the validator to hold the current accepted note if the developer does not wish to accept or reject the note with the next command. This also resets the 5 second escrow timer. (Normally after 5 seconds a note is automatically rejected).

GET BAR CODE READER CONFIGURATION (0X24)

Single byte command causes the validator to return the configuration data for attached bar code readers if there is one present.
Return data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Bar code hardware status. 0x00 = None. 0x01 = Top reader. 0x02 = Bottom reader. 0x03 = Both.	0x03	Both top and bottom barcode readers detected.
2	Enabled status. 0x00 = None. 0x01 = Top. 0x02 = Bottom. 0x03 = Both.	0x03	Both top and bottom barcode readers are enabled.
3	Bar code format. (0x01 = interleaved 2 of 5)	0x00	Not interleaved 2 of 5.
4	Number of characters. Min = 6. Max = 24.	0x0A	10 characters.

SET BAR CODE READER CONFIGURATION (0X23)

Four byte command sets up the validator's bar code reader configuration.
Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x23	The set bar code reader configuration command.
1	Enabled status. 0x00 = None. 0x01 = Top. 0x02 = Bottom. 0x03 = Both.	0x03	Enabling both top and bottom barcode readers.
2	Bar code format. (0x01 = interleaved 2 of 5)	0x00	Not interleaved 2 of 5.

3	Number of characters. Min = 6. Max = 24.	0x0A	10 characters.

GET BAR CODE INHIBIT (0X25)

Single byte command causes validator to return the current bar code/currency inhibit status. This indicates whether the validator can accept only currency, only barcodes, both or neither.
Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	A bit register where bit 0 (lsb) indicates whether currency is enabled. 0 = Enabled. 1 = Disabled. Bit 1 indicates whether the bar code ticket is enabled. 0 = Enabled. 1 = Disabled.	0xFE (11111110 in binary).	Currency is accepted, barcodes are rejected.

SET BAR CODE INHIBIT (0X26)

Two byte command sets bar code/currency inhibits. When the unit is started up or reset, the default is currency enabled, bar code disabled (0xFE).
Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x26	The set bar code inhibit command.
1	A bit register where bit 0 (lsb) indicates whether currency is enabled. 0 = Enabled. 1 = Disabled. Bit 1 indicates whether the bar code ticket is enabled. 0 = Enabled. 1 = Disabled.	0xFE (11111110 in binary).	Setting currency to be accepted, barcodes to be rejected.

GET BAR CODE DATA (0X27)

Single byte command causes validator to return the last valid barcode ticket data.
Return data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Status of the ticket. Single byte. 0x00 = No valid data. 0x01 = Ticket in escrow. 0x02 = Ticket stacked. 0x03 = Ticket rejected.	0x00	No valid data.
2	Length of barcode data. Single byte.	0x02	2 bytes of data follow this length byte.
3*	Barcode data.	0xB5	Example data.
4		0xB6	Example data.

* - The length of this section will vary based on the length of the barcode data.

CONFIGURE BEZEL (0X54)

Four byte command that sets the colour of the bezel to a specified RGB colour. If the validator does not have a bezel that can be modified in this way, 0xF2 (Unknown command) will be returned.

Send Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x54	The configure bezel command.
1	Red byte.	0xFF	Set the red intensity of the bezel.
2	Green byte.	0x00	Set the green intensity of the bezel.
3	Blue byte.	0x00	Set the blue intensity of the bezel.
4	Storage mode. Single byte. 0 = Ram (will return to original on reset). 1 = EEPROM (will persist after reset).	0x00	Stored in Ram. Will return to original when reset.

POLL WITH ACK (0X56)

Single byte command causes the validator to respond to a poll in the same way as normal but specified events will need to be acknowledged by the host using the EVENT ACK before the validator will allow any further note action. If this command is not supported, 0xF2 (Unknown command) will be returned. See appendix for further details about this command.

EVENT ACK (0X57)

Single byte command causes validator to continue with operations after it has been sending a repeating Poll ACK response. See appendix for further details about this command.

9.2 NV11

RESET (0X01)

Single byte command causes the unit to reset.

HOST PROTOCOL VERSION (0X06)

Two byte command sets the unit to report events up to and including those found in the specified protocol version. Please note that the highest protocol version that a unit will support is determined by its firmware. Please see the appendix for more information.

POLL (0X07)

Single byte command instructs the unit to report all the events that have occurred since the last time a poll was sent to the unit. For a more detailed explanation of the poll command and polling the unit, please see section 7 - Polling Devices.

GET SERIAL NUMBER (0X0C)

Single byte command causes the unit to report its unique serial number.
Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Serial number. 4 bytes. Big endian.	0x00	This serial number as a 32 bit integer is 2746699.
2		0x29	
3		0xE9	
4		0x4B	

SYNCHRONISATION COMMAND (0X11)

This single byte command tells the unit that the next sequence ID will be 1. This is always the first command sent to a unit, to prepare it to receive any further commands.

DISABLE (0X09)

This single byte command disables the unit. This means the unit will enter its disabled state and not execute any further commands or perform any other actions. A poll to the unit while in this state will report disabled (0xE8).

ENABLE (0X0A)

Single byte command enables the unit. It will now respond to and execute commands.

DISPLAY ON (0X03)

Single byte command turns on the bezel light when the unit is enabled.

DISPLAY OFF (0X04)

Single byte command turns off the bezel light when the unit is enabled.

REJECT (0X08)

Single byte command causes the validator to reject the current note.

SETUP REQUEST (0X05)

For general information about the setup request see section 6.3.2. Single byte command.

The below table displays the response data of the setup request and provides an example of a real response from a unit with a Euro note dataset.
Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Unit type. Single byte.	0x07	NV11 type.
2	Firmware version. 4 bytes. Each byte represents an ASCII character.	0x00	This gives 0335 when converted to ASCII characters. Formatted it would read 3.35.
3		0x33	
4		0x33	
5		0x35	
6	Country code of validator. 3 bytes. Each byte represents an ASCII character.	0x45	When converted to ASCII characters it reads EUR.
7		0x55	
8		0x52	
9	Value multiplier. 3 bytes. Big endian.	0x00	When converted into a 24 bit integer it will have the value 1.
10		0x00	
11		0x01	
12	Number of channels.	0x04	Four channels.
13*	Channel values. Single byte per channel.	0x05	Five.
14		0x0A	Ten.
15		0x14	Twenty.
16		0x32	Fifty.
17*	Channel security level. Single byte per channel. (Legacy code, now deprecated).	0x02	Level 2.
18		0x02	Level 2.
19		0x02	Level 2.
20		0x02	Level 2.
21*	Real value multiplier. 3 bytes. Big endian.	0x00	When converted into a 24 bit integer it will have the value 100.
22		0x00	
23		0x64	
24*	Protocol version.	0x07	Protocol version 7.
25*	Country codes for each channel. 3 bytes per channel.	0x45	Each channel has the country code EUR when converted to an ASCII character.
26		0x55	
27		0x52	
28		0x45	
29		0x55	
30		0x52	
31		0x45	
32		0x55	
33		0x52	
34		0x45	
35		0x55	
36		0x52	
37*	Channel values for each channel. 4 bytes per channel. Little endian.	0x05	When converted to a 32 bit integer, these values come out at 5, 10, 20, 50.
38		0x00	
39		0x00	
40		0x00	
41		0x0A	
42		0x00	
43		0x00	
44		0x00	
45		0x14	
46		0x00	
47		0x00	
48		0x00	

Copyright Innovative Technology Ltd 2013

49		0x32	
50		0x00	
51		0x00	
52		0x00	

* - These sections' start position and length in the array will vary depending on the number of channels.

UNIT DATA (0X0D)

Single byte command causes the validator to return information about itself. It is similar to the Setup Request command but a more concise version. It is intended for host machines with limited resources. The below table displays the response data of the unit data request and provides an example of a real response from a unit with a Euro note dataset.
Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Unit type. Single byte.	0x00	Note validator type.
2	Firmware version. 4 bytes. Each byte represents an ASCII character.	0x00	This gives 0335 when converted to ASCII characters. Formatted it would read 3.35.
3		0x33	
4		0x33	
5		0x35	
6	Country code of validator. 3 bytes. Each byte represents an ASCII character.	0x45	When converted to ASCII characters it reads EUR.
7		0x55	
8		0x52	
9	Value multiplier. 3 bytes. Big endian.	0x00	When converted into a 24 bit integer it will have the value 1.
10		0x00	
11		0x01	
12	Protocol version.	0x07	Protocol version 7.

CHANNEL VALUE DATA (0X0E)

Single byte command causes the validator to return the number of channels it is using followed by the value of each channel. The below table displays the response data of the channel value data request and provides an example of a real response from a unit with a Euro note dataset.
Return data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Number of channels. Single byte.	0x04	Four channels.
2*	Channel values. Single byte per channel.	0x05	Five.
3		0x0A	Ten.
4		0x14	Twenty.
5		0x32	Fifty.

* - This section's length in the array will vary depending on the number of channels.

LAST REJECT CODE (0X17)

Single byte command causes the validator to report the reason for the last note being rejected.
Return data:

Byte	Description	Response Example	Response Explanation

0	Generic response. Single byte.	0xF0	OK
1	Reject code. Single byte.	0x01	Note length incorrect. For a full list of reject codes see **Appendix**.

HOLD (0X18)

Single byte command causes the validator to hold the current accepted note if the developer does not wish to accept or reject the note with the next command. This also resets the 5 second escrow timer. (Normally after 5 seconds a note is automatically rejected).

ENABLE PAYOUT DEVICE (0X5C)

Single byte command enables storing and paying out notes. Optionally can be sent with an additional byte indicating whether the value of a note is passed with the Note Stored poll response.
Sent data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x5C	The enable payout device command.
1	Optional: A bit register where bit 0 (lsb) indicates whether the value of the note is passed with the Note Stored poll response. 1 = Return the value. 0 = Don't return the value.	0xFE (11111110 in binary).	Setting value to not be reported along with the note stored poll response.

DISABLE PAYOUT DEVICE (0X5B)

Single byte command causes all notes to be routed to the cashbox and payout commands will not be carried out but return the generic response 0xF5 with error code 0x04 meaning that the NV11 payout device is disabled. For more info on generic responses see Appendix.

SET ROUTING (0X3B)

Variable byte command causes the validator to change the recycling status of a note. The first byte of the data will be the route of the note. The next bytes can be different depending on certain factors. If the value reporting type is set to channel, the next byte will be a single byte indicating the channel of the note to change. If the value reporting type is set to value, then the next four bytes will be the 4 byte value of the note.
After this there is 3 country code bytes that are only sent when using protocol version 6+.
Sent Data when using reporting by channel:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x3B	The set routing command.
1	The route of the note. 0 = Recycle. 1 = Don't recycle.	0x00	Setting this note to recycle.
2	The channel of the note.	0x01	Routing note on channel 1.
3	The currency of the note.	0x45	When converted to ASCII characters this will be EUR.
4		0x55	
5		0x52	

Sent Data when using reporting by value:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x3B	The set routing command.
1	The route of the note. 0 = Recycle. 1 = Don't recycle.	0x00	Setting this note to recycle.
2	The value of the note. Little endian.	0xF4	When converted to a 32 bit integer this is 500.
3		0x01	
4		0x00	
5		0x00	
6	The currency of the note.	0x45	When converted to ASCII characters this will be EUR.
7		0x55	
8		0x52	

GET ROUTING (0X3C)

Variable byte command that causes the validator to return the routing for a specific note/channel. This command is the same as the set routing command with regard to the variable length of the command.

Sent Data when using reporting by channel:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x3C	The set routing command.
1	The channel of the note.	0x01	Getting routing for note on channel 1.
2	The currency of the note.	0x45	When converted to ASCII characters this will be EUR.
3		0x55	
4		0x52	

Sent Data when using reporting by value:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x3C	The set routing command.
1	The value of the note.	0xF4	Getting routing for a note with value 500.
2		0x01	
3		0x00	
4		0x00	
6	The currency of the note.	0x45	When converted to ASCII characters this will be EUR.
7		0x55	
8		0x52	

Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Route of note. 0 = Note recycling. 1 = Note not recycling.	0x01	Note not recycling.

EMPTY (0X3F)

Single byte command causes the NV11 to empty all its stored notes to the cashbox.

GET NOTE POSITIONS (0X41)

Single byte command causes the validator to report the number of notes stored and the value of the note in each position. The value reported is either the 4 byte value or the channel number as set by the value reporting type.

Return Data when using reporting by channel:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Number of notes stored. Single byte.	0x02	2 notes stored.
2	Channel of notes stored. 1 byte per channel.	0x01	Both notes in storage are on channel 1.
3		0x01	

Return Data when using reporting by value:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Number of notes stored. Single byte.	0x02	2 notes stored.
2	Value of notes stored. 4 bytes per value.	0xF4	Both notes in storage are of value 500.
3		0x01	
4		0x00	
5		0x00	
6		0xF4	
7		0x01	
8		0x00	
9		0x00	

PAYOUT NOTE (0X42)

Single byte command that causes the validator to payout the next available note stored in the NV11 payout device, this will be the last note that was paid in.

STACK NOTE (0X43)

Single byte command that causes the validator to send the next available note from storage to the cashbox.

SET VALUE REPORTING TYPE (0X45)

Two byte command that changes the way the validator reports the values of notes. There are two options, by channel or by value. When channel is selected the channel number is returned. When value is selected the full 4 byte note value is returned.

Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x45	The set value reporting type command.
1	The type of reporting to use. 0 = By value. 1 = By channel.	0x00	Setting reporting type to report by value.

POLL WITH ACK (0X56)

Single byte command causes the validator to respond to a poll in the same way as normal but specified events will need to be acknowledged by the host using the EVENT ACK before the validator will allow any further note action. If this command is not supported, 0xF2 (Unknown command) will be returned.

EVENT ACK (0X57)

Single byte command causes validator to continue with operations after it has been sending a repeating Poll ACK response.

GET NOTE COUNTERS (0X58)

Single byte command causes validator to report a set of global note counters that track various note statistics. These counters will reset to zero and start again when their maximum value is reached.

Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Number of counters in the set.	0x05	There are 5 counters in the set.
2	Notes the validator has stacked. 4 bytes. Little endian.	0x01	1 note has been stacked.
3		0x00	
4		0x00	
5		0x00	
6	Notes the validator has stored. 4 bytes. Little endian.	0x01	1 note has been stored.
7		0x00	
8		0x00	
9		0x00	
10	Notes the validator has dispensed.	0x00	No notes have been dispensed.
11		0x00	
12		0x00	
13		0x00	
14	Notes the validator has moved from storage to the cashbox.	0x00	No notes have been moved from storage to the cashbox.
15		0x00	
16		0x00	
17		0x00	
18	Notes the validator has rejected.	0x01	1 note has been rejected.
19		0x00	
20		0x00	
21		0x00	

RESET NOTE COUNTERS (0X59)

Single byte command which causes the validator to reset all of its internal note counters to zero.

9.3 SMART PAYOUT

RESET (0X01)

Single byte command causes the unit to reset.

HOST PROTOCOL VERSION (0X06)

Two byte command sets the unit to report events up to and including those found in the specified protocol version. Please note that the highest protocol version that a unit will support is determined by its firmware. Please see the appendix for more information.

POLL (0X07)

Single byte command instructs the unit to report all the events that have occurred since the last time a poll was sent to the unit. For a more detailed explanation of the poll command and polling the unit, please see section 7 - Polling Devices.

GET SERIAL NUMBER (0X0C)

Single byte command causes the unit to report its unique serial number.
Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Serial number. 4 bytes. Big endian.	0x00	This serial number as a 32 bit integer is 2746699.
2		0x29	
3		0xE9	
4		0x4B	

SYNCHRONISATION COMMAND (0X11)

This single byte command tells the unit that the next sequence ID will be 1. This is always the first command sent to a unit, to prepare it to receive any further commands.

DISABLE (0X09)

This single byte command disables the unit. This means the unit will enter its disabled state and not execute any further commands or perform any other actions. A poll to the unit while in this state will report disabled (0xE8).

ENABLE (0X0A)

Single byte command enables the unit. It will now respond to and execute commands.

DISPLAY ON (0X03)

Single byte command turns on the bezel light when the unit is enabled.

DISPLAY OFF (0X04)

Single byte command turns off the bezel light when the unit is enabled.

REJECT (0X08)

Single byte command causes the validator to reject the current note.

SETUP REQUEST (0X05)

For general information about the setup request see section 6.3.2. Single byte command.

Copyright Innovative Technology Ltd 2013

The below table displays the response data of the setup request and provides an example of a real response from a unit with a Euro note dataset.
Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Unit type. Single byte.	0x06	SMART Payout type.
2	Firmware version. 4 bytes. Each byte represents an ASCII character.	0x00	This gives 0411 when converted to ASCII characters. Formatted it would read 4.11.
3		0x34	
4		0x31	
5		0x31	
6	Country code of validator. 3 bytes. Each byte represents an ASCII character.	0x45	When converted to ASCII characters it reads EUR.
7		0x55	
8		0x52	
9	Value multiplier. 3 bytes. Big endian.	0x00	When converted into a 24 bit integer it will have the value 1.
10		0x00	
11		0x01	
12	Number of channels.	0x04	Four channels.
13*	Channel values. Single byte per channel.	0x05	Five.
14		0x0A	Ten.
15		0x14	Twenty.
16		0x32	Fifty.
17*	Channel security level. Single byte per channel. (Legacy code, now deprecated).	0x02	Level 2.
18		0x02	Level 2.
19		0x02	Level 2.
20		0x02	Level 2.
21*	Real value multiplier. 3 bytes. Big endian.	0x00	When converted into a 24 bit integer it will have the value 100.
22		0x00	
23		0x64	
24*	Protocol version.	0x07	Protocol version 7.
25*	Country codes for each channel. 3 bytes per channel.	0x45	Each channel has the country code EUR when converted to an ASCII character.
26		0x55	
27		0x52	
28		0x45	
29		0x55	
30		0x52	
31		0x45	
32		0x55	
33		0x52	
34		0x45	
35		0x55	
36		0x52	
37*	Channel values for each channel. 4 bytes per channel. Little endian.	0x05	When converted to a 32 bit integer, these values come out at 5, 10, 20, 50.
38		0x00	
39		0x00	
40		0x00	
41		0x0A	
42		0x00	
43		0x00	
44		0x00	
45		0x14	
46		0x00	
47		0x00	
48		0x00	

49		0x32	
50		0x00	
51		0x00	
52		0x00	

* - These sections' start position and length in the array will vary depending on the number of channels.

UNIT DATA (0X0D)

Single byte command causes the validator to return information about itself. It is similar to the Setup Request command but a more concise version. It is intended for host machines with limited resources. The below table displays the response data of the unit data request and provides an example of a real response from a unit with a Euro note dataset.
Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Unit type. Single byte.	0x00	Note validator type.
2	Firmware version. 4 bytes. Each byte represents an ASCII character.	0x00	This gives 0411 when converted to ASCII characters. Formatted it would read 4.11.
3		0x34	
4		0x31	
5		0x31	
6	Country code of validator. 3 bytes. Each byte represents an ASCII character.	0x45	When converted to ASCII characters it reads EUR.
7		0x55	
8		0x52	
9	Value multiplier. 3 bytes. Big endian.	0x00	When converted into a 24 bit integer it will have the value 1.
10		0x00	
11		0x01	
12	Protocol version.	0x07	Protocol version 7.

CHANNEL VALUE DATA (0X0E)

Single byte command causes the validator to return the number of channels it is using followed by the currency and value of each channel. The below table displays the response data of the channel value data request and provides an example of a real response from a unit with a Euro note dataset.
Return data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Number of channels. Single byte.	0x02	Two channels.
2*	Channel values. Single byte per channel. Legacy code. No longer needed.	0x01	1.
3		0x02	2.
4*	Currency of channel, 3 bytes.	0x45	Converted to ASCII characters this is EUR.
5		0x55	
6		0x52	
7		0x45	Converted to ASCII characters this is EUR.
8		0x55	
9		0x52	
18*	Value of channel, 4 bytes. Little endian.	0x05	This value is 5.
19		0x00	
20		0x00	
21		0x00	
22		0x0A	This value is 10.

23		0x00	
24		0x00	
25		0x00	

* - These sections' length in the array will vary depending on the number of channels.

LAST REJECT CODE (0X17)

Single byte command causes the validator to report the reason for the last note being rejected.
Return data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Reject code. Single byte.	0x01	Note length incorrect. For a full list of reject codes **Appendix**.

HOLD (0X18)

Single byte command causes the validator to hold the current accepted note if the developer does not wish to accept or reject the note with the next command. This also resets the 5 second escrow timer. (Normally after 5 seconds a note is automatically rejected).

GET BAR CODE READER CONFIGURATION (0X24)

Single byte command causes the validator to return the configuration data for attached bar code readers if there is one present.
Return data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Bar code hardware status. 0x00 = None. 0x01 = Top reader. 0x02 = Bottom reader. 0x03 = Both.	0x03	Both top and bottom barcode readers detected.
2	Enabled status. 0x00 = None. 0x01 = Top. 0x02 = Bottom. 0x03 = Both.	0x03	Both top and bottom barcode readers are enabled.
3	Bar code format. (0x01 = interleaved 2 of 5)	0x00	Not interleaved 2 of 5.
4	Number of characters. Min = 6. Max = 24.	0x0A	10 characters.

SET BAR CODE READER CONFIGURATION (0X23)

Four byte command sets up the validator's bar code reader configuration.
Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x23	The set bar code reader configuration command.

1	Enabled status. 0x00 = None. 0x01 = Top. 0x02 = Bottom. 0x03 = Both.	0x03	Enabling both top and bottom barcode readers.
2	Bar code format. (0x01 = interleaved 2 of 5)	0x00	Not interleaved 2 of 5.
3	Number of characters. Min = 6. Max = 24.	0x0A	10 characters.

GET BAR CODE INHIBIT (0X25)

Single byte command causes validator to return the current bar code/currency inhibit status. This indicates whether the validator can accept only currency, only barcodes, both or neither.
Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	A bit register where bit 0 (lsb) indicates whether currency is enabled. 0 = Enabled. 1 = Disabled. Bit 1 indicates whether the bar code ticket is enabled. 0 = Enabled. 1 = Disabled.	0xFE (11111110 in binary).	Currency is accepted, barcodes are rejected.

SET BAR CODE INHIBIT (0X26)

Two byte command sets bar code/currency inhibits. When the unit is started up or reset, the default is currency enabled, bar code disabled (0xFE).
Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x26	The set bar code inhibit command.
1	A bit register where bit 0 (lsb) indicates whether currency is enabled. 0 = Enabled. 1 = Disabled. Bit 1 indicates whether the bar code ticket is enabled. 0 = Enabled. 1 = Disabled.	0xFE (11111110 in binary).	Setting currency to be accepted, barcodes to be rejected.

GET BAR CODE DATA (0X27)

Single byte command causes validator to return the last valid barcode ticket data.
Return data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK

1	Status of the ticket. Single byte. 0x00 = No valid data. 0x01 = Ticket in escrow. 0x02 = Ticket stacked. 0x03 = Ticket rejected.	0x00	No valid data.
2	Length of barcode data. Single byte.	0x02	2 bytes of data follow this length byte.
3*	Barcode data.	0xB5	Example data.
4		0xB6	Example data.

* - The length of this section will vary based on the length of the barcode data.

ENABLE PAYOUT DEVICE (0X5C)

Single byte command enables storing and paying out notes.

DISABLE PAYOUT DEVICE (0X5B)

Single byte command causes all notes to be routed to the cashbox and payout commands will not be carried out but instead return the generic response 0xF5 with error code 0x04 meaning that the Payout device is disabled. For more info on generic responses see Appendix.

SET ROUTING (0X3B)

Nine byte command causes the validator to change the recycling status of a note. The first byte of the data will be the route of the note. Then there are 4 bytes holding the value of the note, finally there is 3 country code bytes.

Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x3B	The set routing command.
1	The route of the note. 0 = Recycle. 1 = Don't recycle.	0x00	Setting this note to recycle.
2	The value of the note. Little endian.	0xF4	When converted to a 32 bit integer this is 500.
3		0x01	
4		0x00	
5		0x00	
6	The currency of the note.	0x45	When converted to ASCII characters this will be EUR.
7		0x55	
8		0x52	

GET ROUTING (0X3C)

Nine byte command that causes the validator to return the routing for a specific note.

Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x3C	The set routing command.
1	The value of the note.	0xF4	Getting routing for a note with value 500.
2		0x01	
3		0x00	
4		0x00	
6	The currency of the note.	0x45	When converted to ASCII characters this will be EUR.
7		0x55	
8		0x52	

Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Route of note. 0 = Note recycling. 1 = Note not recycling.	0x01	Note not recycling.

EMPTY (0X3F)

Single byte command causes the SMART Payout to empty all its stored notes to the cashbox.

PAYOUT AMOUNT (0X33)

Variable byte command that instructs the payout device to payout a specified amount. The developer can specify whether the payout is a “real” payout or a “test” payout. This can be useful as it allows the developer to find out whether a payout could be made without actually making the payout. This is done using an additional byte at the end of the standard data. The example below demonstrates paying out €15.00.

Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x33	The payout amount command.
1	The value of the payout.	0xDC	Paying out 1500 (penny value).
2		0x05	
3		0x00	
4		0x00	
6	The currency of the note.	0x45	When converted to ASCII characters this will be EUR.
7		0x55	
8		0x52	
9	The payout option, test or real. 0x58 = Real. 0x19 = Test.	0x58	Real payout.

GET NOTE AMOUNT (0X35)

Variable byte command that causes the validator to report the amount of notes stored of a specified denomination in the payout unit.

Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x35	The get note amount command.
1	The value of the note.	0xF4	Finding out how many notes of value 500.
2		0x01	
3		0x00	
4		0x00	
6	The currency of the note.	0x45	When converted to ASCII characters this will be EUR.
7		0x55	
8		0x52	

Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Two byte value of the	0x01	There is one note stored.

2	amount of notes of that denomination and currency stored. Little endian.	0x00	

GET ALL LEVELS (0X22)

Single byte command that causes the SMART Payout to report the amount of notes stored for all denominations.
Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x22	Get all levels command.

Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response.	0xF0	OK
1	Number of denominations	0x03	3 denominations
2	Two byte count of denom 1 coins stored. Little endian.	0x05	5 notes stored
3		0x00	
4	Four byte value of denom 1. Little endian.	0xF4	Value 500
5		0x01	
6		0x00	
7		0x00	
8	3 byte currency code of denom 1.	0x45	When converted to ASCII characters this will be EUR.
9		0x55	
10		0x52	
11	Two byte count of denom 2 coins stored. Little endian.	0x0E	15 notes stored
12		0x00	
13	Four byte value of denom 2. Little endian.	0xE8	Value 1000
14		0x03	
15		0x00	
16		0x00	
17	3 byte currency code of denom 2.	0x45	When converted to ASCII characters this will be EUR.
18		0x55	
19		0x52	
20	Two byte count of denom 3 coins stored. Little endian.	0x05	5 notes stored
21		0x00	
22	Four byte value of denom 3. Little endian.	0xD0	Value 2000
23		0x07	
24		0x00	
25		0x00	
26	3 byte currency code of denom 3.	0x45	When converted to ASCII characters this will be EUR.
27		0x55	
28		0x52	

HALT PAYOUT (0X38)

Single byte command that causes the current payout to stop.

FLOAT AMOUNT (0X3D)

Variable byte command that causes the validator to keep a set amount “floating” in the payout and specifies a minimum payout value. In a similar way to the Payout Amount command there is an option byte at the end to make a “real” float or a “test” float.
Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x35	The get note amount command.

Copyright Innovative Technology Ltd 2013

1	The minimum payout that should be available in the Payout.	0xF4	Minimum payout set to 500 (penny value).
2		0x01	
3		0x00	
4		0x00	
6	The float amount that should be left in the Payout.	0x10	Leaving 10000 (penny value) in the payout.
7		0x27	
8		0x00	
9		0x00	
10	The currency of the float.	0x45	When converted to ASCII characters this is EUR.
11		0x55	
12		0x52	
13	The float option, test or real. 0x58 = Real. 0x19 = Test.	0x58	Performing a real float.

GET MINIMUM PAYOUT (0X3E)

Variable byte command causes the validator to report its current minimum payout of a specific currency.

Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x3E	The get minimum payout command.
1	The currency of the minimum payout.	0x45	Converted to ASCII characters this would be EUR.
2		0x55	
3		0x52	

Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Four byte value of the minimum payout. Little endian.	0xF4	The minimum payment is 500 (penny value).
2		0x01	
3		0x00	
4		0x00	

PAYOUT BY DENOMINATION (0X46)

Variable byte command that instructs the validator to payout the requested number of a denomination of a note. This differs from a standard payout command (0x33) in that the developer specifies exactly which notes to payout. In the standard payout, the validator decides, based on the total amount the developer sends it.

The following tables use the example of a developer wanting to payout 5 X 5.00 EUR notes and 5 X 10.00 EUR notes.

Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x46	The payout by denomination command.
1	Number of different denominations to payout.	0x02	Two denominations required.
2*	Number of the first denomination to payout. Two bytes.	0x05	Payout 5 of this denomination.
3		0x00	
4	Value of this denomination.	0xF4	This denomination's value

5	4 bytes.	0x01	is 500 (penny value).
6		0x00	
7		0x00	
8	Country code of this denomination.	0x45	The country code when converted to ASCII characters is EUR.
9		0x55	
10		0x52	
11	Number of the second denomination to payout.	0x05	Payout 5 of this denomination.
12		0x00	
13	Value of this denomination.	0xE8	This denomination's value is 1000 (penny value).
14		0x03	
15		0x00	
16		0x00	
17	Country code of this denomination.	0x45	The country code when converted to ASCII characters is EUR.
18		0x55	
19		0x52	
20	The payout option. 58 = real payout. 19 = test payout.	0x58	Perform a real payout.

* - The length of this section will vary based on the number of denominations being paid out.

FLOAT BY DENOMINATION (0X44)

Variable byte command that instructs the validator to float individual quantities of a denomination in the SMART payout. It follows a similar format to the Payout by Denomination command.

The following tables use the example of a developer wanting to float 5 X 5.00 EUR notes and 5 X 10.00 EUR notes.
Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x44	The float by denomination command.
1	Number of different denominations to float.	0x02	Two denominations required.
2*	Number of the first denomination to float. Two bytes.	0x05	Float 5 of this denomination.
3		0x00	
4	Value of this denomination. 4 bytes.	0xF4	This denomination's value is 500 (penny value).
5		0x01	
6		0x00	
7		0x00	
8	Country code of this denomination.	0x45	The country code when converted to ASCII characters is EUR.
9		0x55	
10		0x52	
11*	Number of the second denomination to float.	0x05	Float 5 of this denomination.
12		0x00	
13	Value of this denomination.	0xE8	This denomination's value is 1000 (penny value).
14		0x03	
15		0x00	
16		0x00	
17	Country code of this denomination.	0x45	The country code when converted to ASCII characters is EUR.
18		0x55	
19		0x52	

20	The payout option. 58 = real float. 19 = test float.	0x58	Perform a real float.

* - The length of this section will vary based on the number of denominations being floated.

SMART EMPTY (0X52)

Single byte command that causes the validator to empty all its stored notes to the cashbox and also keep a count of the value emptied. This information can be retrieved using the cashbox payout operation data command once the payout is empty.

CASHBOX PAYOUT OPERATION DATA (0X53)

Single byte command that instructs the validator to return the amount emptied from the payout to the cashbox in the last dispense, SMART empty or float operation.
Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Number of denominations in this response. Single byte.	0x02	Two denominations.
2*	Number of the first denomination moved. Two bytes.	0x01	Moved 1 of this denomination.
3		0x00	
4	Value of the denomination moved. 4 bytes.	0xF4	The value of this denomination is 500 (penny value).
5		0x01	
6		0x00	
7		0x00	
8	Country code of the denomination moved.	0x45	When converted to ASCII characters this is EUR.
9		0x55	
10		0x52	
11	Number of the first denomination moved. Two bytes.	0x00	Moved none of this denomination.
12		0x00	
13	Value of the denomination moved. 4 bytes.	0xE8	The value of this denomination is 500 (penny value).
14		0x03	
15		0x00	
16		0x00	
17	Country code of the denomination moved.	0x45	When converted to ASCII characters this is EUR.
18		0x55	
19		0x52	
20**	Number of notes that were moved but not recognised. 4 bytes.	0x00	No notes were moved without being recognised.
21		0x00	
22		0x00	
23		0x00	

* - The length of this section will vary based on the number of denominations being reported.
** - The position of this section will vary based on the number of denominations reported.

POLL WITH ACK (0X56)

Single byte command causes the validator to respond to a poll in the same way as normal but specified events will need to be acknowledged by the host using the EVENT ACK before the validator will allow any further note action. If this command is not supported, 0xF2 (Unknown command) will be returned.

EVENT ACK (0X57)

Single byte command causes validator to continue with operations after it has been sending a repeating Poll ACK response.

GET NOTE COUNTERS (0X58)

Single byte command causes validator to report a set of global note counters that track various note statistics. These counters will reset to zero and start again when their maximum value is reached.
Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Number of counters in the set.	0x05	There are 5 counters in the set.
2	Notes the validator has stacked. 4 bytes. Little endian.	0x01	1 note has been stacked.
3		0x00	
4		0x00	
5		0x00	
6	Notes the validator has stored. 4 bytes. Little endian.	0x01	1 note has been stored.
7		0x00	
8		0x00	
9		0x00	
10	Notes the validator has dispensed.	0x00	No notes have been dispensed.
11		0x00	
12		0x00	
13		0x00	
14	Notes the validator has moved from storage to the cashbox.	0x00	No notes have been moved from storage to the cashbox.
15		0x00	
16		0x00	
17		0x00	
18	Notes the validator has rejected.	0x01	1 note has been rejected.
19		0x00	
20		0x00	
21		0x00	

RESET NOTE COUNTERS (0X59)

Single byte command which causes the validator to reset all of its internal note counters to zero.

SET REFILL MODE (0X30)

Five or six byte command sequence which causes the payout to change or report its refill mode. By default if a note is inserted which the firmware determines is unsuitable for storage, it is sent to the cashbox instead. If the refill mode is active then the note is rejected from the front of the validator to make refilling more convenient for the user. This command is sent as a sequence of bytes and is only available in firmware 4.10+.
Sent Data:

Byte	Description	Command Example	Command Explanation
0	The first command byte of the sequence.	0x30	Set refill mode command byte.
1	The next three bytes are the command sequence bytes. They are always the same.	0x05	Set refill mode command sequence.
2		0x81	
3		0x10	

4	This byte is the read/write option byte. 0x11 = write 0x01 = read	0x11	This will set the mode to write.
5	The option byte, sets refill mode either on or off. 0x00 = no refill mode. 0x01 = refill mode. If the previous byte is read (0x01) then the response will indicate the validator's current refill mode and this byte does not need to be included.	0x01	Set validator to refill mode.

Return Data (if byte 4 in the above command is set to read (0x01)):

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Refill mode. 0x00 – no refill mode. 0x01 – refill mode.	0x01	Refill mode is on.

9.4 SMART HOPPER

RESET (0X01)

Single byte command causes the unit to reset.

HOST PROTOCOL VERSION (0X06)

Two byte command sets the unit to report events up to and including those found in the specified protocol version. Please note that the highest protocol version that a unit will support is determined by its firmware. Please see the appendix for more information.

POLL (0X07)

Single byte command instructs the unit to report all the events that have occurred since the last time a poll was sent to the unit. For a more detailed explanation of the poll command and polling the unit, please see section 7 - Polling Devices.

GET SERIAL NUMBER (0X0C)

Single byte command causes the unit to report its unique serial number.
Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Serial number. 4 bytes. Big endian.	0x00	This serial number as a 32 bit integer is 2746699.
2		0x29	
3		0xE9	
4		0x4B	

SYNCHRONISATION COMMAND (0X11)

This single byte command tells the unit that the next sequence ID will be 1. This is always the first command sent to a unit, to prepare it to receive any further commands.

DISABLE (0X09)

This single byte command disables the unit. This means the unit will enter its disabled state and not execute any further commands or perform any other actions. A poll to the unit while in this state will report disabled (0xE8).

ENABLE (0X0A)

Single byte command enables the unit. It will now respond to and execute commands.

SETUP REQUEST (0X05)

For general information about the setup request see section 6.3.2. Single byte command.
The below table displays the response data of the setup request and provides an example of a real response from a SMART Hopper with a Euro coin dataset.
Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Unit type. Single byte.	0x03	SMART Hopper type.
2	Firmware version. 4 bytes.	0x00	This reads as 0612 when converted to ASCII characters. Formatted it reads 6.12.
3		0x36	
4		0x31	
5		0x32	

Copyright Innovative Technology Ltd 2013

6	Country code of validator. 3 bytes. Each byte represents an ASCII character.	0x45	When converted to ASCII characters it reads EUR.
7		0x55	
8		0x52	
9	Protocol version. Single byte.	0x07	Protocol version 7.
10	Number of channels.	0x07	Seven channels.
11*	Value of channels. Two bytes per channel. Little endian.	0x02	Value is 2 (penny value).
12		0x00	
13		0x05	Value is 5 (penny value).
14		0x00	
15		0x0A	Value is 10 (penny value).
16		0x00	
17		0x14	Value is 20 (penny value).
18		0x00	
19		0x32	Value is 50 (penny value).
20		0x00	
21		0x64	Value is 100 (penny value).
22		0x00	
23		0xC8	Value is 200 (penny value).
24		0x00	
25*	Country codes of channels. 3 bytes per channel.	0x45	Converted to ASCII characters this reads EUR.
26		0x55	
27		0x52	
28		0x45	Converted to ASCII characters this reads EUR.
29		0x55	
30		0x52	
31		0x45	Converted to ASCII characters this reads EUR.
32		0x55	
33		0x52	
34		0x45	Converted to ASCII characters this reads EUR.
35		0x55	
36		0x52	
37		0x45	Converted to ASCII characters this reads EUR.
38		0x55	
39		0x52	
40		0x45	Converted to ASCII characters this reads EUR.
41		0x55	
42		0x52	
43		0x45	Converted to ASCII characters this reads EUR.
44		0x55	
45		0x52	

* - The positions and/or lengths of these sections depend on the number of channels being used by the validator.

SET ROUTING (0X3B)

Eight byte command causes the SMART Hopper to change the recycling status of a coin. The first byte of the data will be the route of the coin. Then there 4 bytes holding the value of the coin, finally there is 3 country code bytes.

Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x3B	The set routing command.
1	The route of the coin. 0 = Recycle. 1 = Don't recycle.	0x00	Setting this coin to recycle.

2	The value of the coin. Little endian.	0xC8	When converted to a 32 bit integer this is 200.
3		0x00	
4		0x00	
5		0x00	
6	The currency of the coin.	0x45	When converted to ASCII characters this will be EUR.
7		0x55	
8		0x52	

GET ROUTING (0X3C)

Nine byte command that causes the validator to return the routing for a specific coin.
Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x3C	The set routing command.
1	The value of the coin.	0xC8	Getting routing for a coin with value 200.
2		0x00	
3		0x00	
4		0x00	
6	The currency of the coin.	0x45	When converted to ASCII characters this will be EUR.
7		0x55	
8		0x52	

PAYOUT AMOUNT (0X33)

Ten byte command that instructs the Hopper to payout a specified amount. The developer can specify whether the payout is a “real” payout or a “test” payout. This can be useful as it allows the developer to find out whether a payout could be made without actually making the payout. This is done using an additional byte at the end of the standard data. The example below demonstrates paying out €12.46.
Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x33	The payout amount command.
1	The value of the payout.	0xDE	Paying out 1246 (penny value).
2		0x04	
3		0x00	
4		0x00	
6	The currency of the note.	0x45	When converted to ASCII characters this will be EUR.
7		0x55	
8		0x52	
9	The payout option, test or real. 0x58 = Real. 0x19 = Test.	0x58	Real payout.

GET COIN AMOUNT (0X35)

Variable byte command that causes the SMART Hopper to report the amount of coins stored of a specified denomination.
Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x35	The get coin amount command.
1	The value of the coin.	0xC8	Finding out how many coins of value 200.
2		0x00	
3		0x00	
4		0x00	

6	The currency of the coin.	0x45	When converted to ASCII characters this will be EUR.
7		0x55	
8		0x52	

Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Two byte value of the amount of coins of that denomination and currency stored. Little endian.	0x32	There are fifty coins stored.
2		0x00	

GET ALL LEVELS (0X22)

Single byte command that causes the SMART Hopper to report the amount of coins stored for all denominations.
Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x22	Get all levels command.

Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response.	0xF0	OK
1	Number of denominations	0x03	3
2	Two byte count of denom 1 coins stored. Little endian.	0x2E	46 coins stored
3		0x00	
4	Four byte value of denom 1. Little endian.	0x32	Value 50
5		0x00	
6		0x00	
7		0x00	
8	3 byte currency code of denom 1.	0x45	When converted to ASCII characters this will be EUR.
9		0x55	
10		0x52	
11	Two byte count of denom 2 coins stored. Little endian.	0x4B	75 coins stored
12		0x00	
13	Four byte value of denom 2. Little endian.	0x32	Value 100
14		0x00	
15		0x00	
16		0x00	
17	3 byte currency code of denom 2.	0x45	When converted to ASCII characters this will be EUR.
18		0x55	
19		0x52	
20	Two byte count of denom 3 coins stored. Little endian.	0x18	24 coins stored
21		0x00	
22	Four byte value of denom 3. Little endian.	0x32	Value 200
23		0x00	
24		0x00	
25		0x00	
26	3 byte currency code of denom 3.	0x45	When converted to ASCII characters this will be EUR.
27		0x55	
28		0x52	

SET COIN AMOUNT (0X34)

Nine byte command that increases the level of a particular denomination of coin in the SMART Hopper by a specified amount. Note that although the start of this command name is "set", it actually increments the level, not sets it.

Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x34	The set coin amount command.
1	The number of coins to add to the level of this denomination. 2 bytes.	0x0A	Adding 10 coins.
2		0x00	
3	The value of the denomination. 4 bytes.	0x32	The denomination value is 50 (penny value).
4		0x00	
5		0x00	
6		0x00	
7	The country code of the denomination. 3 bytes.	0x45	The country code when converted to ASCII characters is EUR.
8		0x55	
9		0x52	

HALT PAYOUT (0X38)

Single byte command that halts the current payout.

FLOAT AMOUNT (0X3D)

Fourteen byte command that causes the validator to keep a set amount "floating" in the SMART Hopper and specifies a minimum payout value. In a similar way to the Payout Amount command there is an option byte at the end to make a "real" float or a "test" float.

Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x35	The get note amount command.
1	The minimum payout that should be available to payout in the SMART Hopper.	0x32	Minimum payout set to 50 (penny value).
2		0x00	
3		0x00	
4		0x00	
6	The float amount that should be left in the SMART Hopper.	0x10	Leaving 10000 (penny value) in the payout.
7		0x27	
8		0x00	
9		0x00	
10	The currency of the float.	0x45	When converted to ASCII characters this is EUR.
11		0x55	
12		0x52	
13	The float option, test or real. 0x58 = Real. 0x19 = Test.	0x58	Performing a real float.

GET MINIMUM PAYOUT (0X3E)

Variable byte command causes the SMART Hopper to report its current minimum payout of a specific currency.

Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x3E	The get minimum payout command.
1	The currency of the	0x45	Converted to ASCII

2	minimum payout.	0x55	characters this would be EUR.
3		0x52	

Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Four byte value of the minimum payout. Little endian.	0x32	The minimum payment is 50 (penny value).
2		0x00	
3		0x00	
4		0x00	

SET COIN MECH INHIBITS (0X40)

Seven byte command causes the SMART Hopper to disable or enable acceptance of individual coin denominations by an attached coin mechanism. If this command is sent to a SMART Hopper with no coin mechanism attached then it will return the generic response 0xF3 (Wrong Parameters) for more info on generic responses see Appendix.

Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x40	The set coin mech inhibits command.
1	Inhibition status of coin. 0 = inhibited 1 = uninhibited	0x00	Inhibiting this coin.
2	The value of the coin. Two bytes.	0x02	The coin value is 2 (penny value).
3		0x00	
4	The country code of the coin.	0x45	When converted to ASCII characters this reads EUR.
5		0x55	
6		0x52	

PAYOUT BY DENOMINATION (0X46)

Variable byte command that instructs the SMART Hopper to payout the requested number of a denomination of coin. This differs from a standard payout command (0x33) in that the developer specifies exactly which coins to payout. In the standard payout, the SMART Hopper decides which coins to payout, based on the total amount the developer sends it.

The following tables use the example of a developer wanting to payout 5 X 0.50 EUR coins and 5 X 1.00 EUR coins.

Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x46	The payout by denomination command.
1	Number of different denominations to payout.	0x02	Two denominations required.
2*	Number of the first denomination to payout. Two bytes.	0x05	Payout 5 of this denomination.
3		0x00	
4	Value of this denomination. 4 bytes.	0x32	This denomination’s value is 50 (penny value).
5		0x00	
6		0x00	
7		0x00	
8	Country code of this denomination.	0x45	The country code when converted to ASCII characters is EUR.
9		0x55	
10		0x52	

Copyright Innovative Technology Ltd 2013

11	Number of the second denomination to payout.	0x05	Payout 5 of this denomination.
12		0x00	
13	Value of this denomination.	0x64	This denomination's value is 100 (penny value).
14		0x00	
15		0x00	
16		0x00	
17	Country code of this denomination.	0x45	The country code when converted to ASCII characters is EUR.
18		0x55	
19		0x52	
20	The payout option. 58 = real payout. 19 = test payout.	0x58	Perform a real payout.

* - The length of this section will vary based on the number of denominations being paid out.

FLOAT BY DENOMINATION (0X44)

Variable byte command that instructs the SMART Hopper to float the requested number of a denomination of coin.

The following tables use the example of a developer wanting to float 5 X 0.50 EUR coins and 5 X 1.00 EUR coins.

Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x44	The float by denomination command.
1	Number of different denominations to float.	0x02	Two denominations required.
2*	Number of the first denomination to float. Two bytes.	0x05	Float 5 of this denomination.
3		0x00	
4	Value of this denomination. 4 bytes.	0x32	This denomination's value is 50 (penny value).
5		0x00	
6		0x00	
7		0x00	
8	Country code of this denomination.	0x45	The country code when converted to ASCII characters is EUR.
9		0x55	
10		0x52	
11	Number of the second denomination to float. Two bytes.	0x05	Float 5 of this denomination.
12		0x00	
13	Value of this denomination.	0x64	This denomination's value is 100 (penny value).
14		0x00	
15		0x00	
16		0x00	
17	Country code of this denomination.	0x45	The country code when converted to ASCII characters is EUR.
18		0x55	
19		0x52	
20	The float option. 58 = real float. 19 = test float.	0x58	Perform a real float.

* - The length from this byte onwards will vary based on the number of denominations being floated.

SET COMMAND CALIBRATION (0X47)

Two byte command that causes the SMART Hopper to set its calibration mode to either “auto calibration” or “command calibration”. Auto calibration is the default mode and will run the calibration at intervals determined by the firmware. In command calibration mode the SMART Hopper will only run its calibration sequence when commanded by the host. If the host does not send a calibration command within the calibration period, the SMART Hopper will respond with a calibration fail event until the Run Command Calibration command is sent.
Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x47	The set command calibration command.
1	The calibration mode. 0 = Auto. 1 = Command.	0x00	Setting the calibration mode to auto.

RUN COMMAND CALIBRATION (0X48)

Depreciated and no longer used.

EMPTY ALL (0X3F)

Single byte command that causes the SMART Hopper to empty all its stored coins to the cashbox.

SET OPTIONS (0X50)

Two byte command that sets various options on the SMART Hopper. These options are volatile and will not persist in memory after a reset. This command is only available in firmware 6.04+.

The command works by sending two bytes which act as bit registers after the command byte. The first bit register looks as follows:

REGISTER 1							
Not used.	Not used.	Not used.	Not used.	Cashbox Pay.	Motor Speed.	Level Check.	Pay Mode.

Pay Mode – This can either be set to free pay (1) or split by highest value (0). When free pay is selected the Hopper pays out the first coins that pass its discriminator system if it fits into the current payout value and will leave enough of other coins to payout the remaining value. This gives faster payouts but could result in lots of small denomination coins being paid out. This is the default state after the unit is reset.
The other pay mode is split by highest value where the Hopper attempts to pay from the highest value coin it can, this will payout the minimum number of coins possible.

Level Check – When the level check is enabled (1) the Hopper will check the levels of coins before it tries to make a payout. When this is disabled the Hopper will attempt to payout any amount without checking the levels first.

Motor Speed – Payouts will run at a lower motor speed when this is set to 0. When set to 1 the motor runs at max speed.

Cashbox Pay – This works in conjunction with the pay mode bit. If this bit is 0 then the pay modes will be as described in the pay mode bit. When this bit is 1 then coins routed to the cashbox will be used in coins paid out of the front, if they fit in the current payout request.
This table shows the relation:

Cashbox Pay Bit	Pay Mode Bit	Pay Mode
0	0	Split by highest value

0	1	Free pay
1	0	Split by highest, use cashbox coins in split.
0	1	Free pay, use cashbox coins in pay.

The second bit register is not used at present and should be set to 0x00.

Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x50	The set options command.
1	Register 1.	0x06	Setting cashbox pay to false, motor speed to highest, level check to true and pay mode to split by highest value. (00000110).
2	Register 2.	0x00	Not used.

GET OPTIONS (0X51)

Single byte command that instructs the SMART Hopper to return the two option bytes described in the set options command.
Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Register 1.	0x06	Cashbox pay is false, motor speed is highest, level check is true and pay mode to split by highest value is set. (00000110).
2	Register 2	0x00	Not used.

COIN MECH GLOBAL INHIBIT (0X49)

Two byte command that causes a coin acceptor attached to the SMART Hopper to inhibit or un-inhibit all of its channels (effectively disabling or enabling the coin mechanism). Please note, if this command is sent to a SMART Hopper without a coin mechanism attached, it will return the generic response 0xF3 (wrong parameters).
Sent Data:

Byte	Description	Command Example	Command Explanation
0	Command byte.	0x50	The set options command.
1	Inhibit mode. 0 = All inhibited. 1 = None inhibited.	0x01	Enable all channels.

SMART EMPTY (0X52)

Single byte command that causes the validator to empty all its stored coins to the cashbox and also keep a count of the value emptied. This information can be retrieved using the Cashbox Payout Operation Data command once the SMART Hopper is empty.

CASHBOX PAYOUT OPERATION DATA (0X53)

Single byte command that instructs the SMART Hopper to return the amount emptied from the payout to the cashbox in the last dispense, SMART empty or float operation.

Copyright Innovative Technology Ltd 2013

Return Data:

Byte	Description	Response Example	Response Explanation
0	Generic response. Single byte.	0xF0	OK
1	Number of denominations in this response. Single byte.	0x02	Two denominations.
2*	Number of the first denomination moved. Two bytes.	0x0A	Moved 10 of this denomination.
3		0x00	
4	Value of the denomination moved. 4 bytes.	0x32	The value of this denomination is 50 (penny value).
5		0x00	
6		0x00	
7		0x00	
8	Country code of the denomination moved.	0x45	When converted to ASCII characters this is EUR.
9		0x55	
10		0x52	
11	Number of the first denomination moved. Two bytes.	0x00	Moved none of this denomination.
12		0x00	
13	Value of the denomination moved. 4 bytes.	0x64	The value of this denomination is 100 (penny value).
14		0x00	
15		0x00	
16		0x00	
17	Country code of the denomination moved.	0x45	When converted to ASCII characters this is EUR.
18		0x55	
19		0x52	
20**	Number of notes that were moved but not recognised. 4 bytes.	0x00	No notes were moved without being recognised.
21		0x00	
22		0x00	
23		0x00	

POLL WITH ACK (0X56)

Single byte command causes the SMART Hopper to respond to a poll in the same way as normal but specified events will need to be acknowledged by the host using the EVENT ACK before the unit will allow any further note action. If this command is not supported, 0xF2 (Unknown command) will be returned.

EVENT ACK (0X57)

Single byte command causes SMART Hopper to continue with operations after it has been sending a repeating Poll ACK response.

COIN MECH OPTIONS (0X5A)

The host can set the following options for the Smart Hopper. These options do not persist in memory and after a reset they will go to their default values.

REG 0	Parameter
Bit 0	Coin Mech error events 1 = ccTalk format, 0 = Coin mech jam and Coin return mech open only
Bit 1	Not used – set to 0
Bit 2	Not used – set to 0
Bit 3	Not used – set to 0

Bit 4	Not used – set to 0
Bit 5	Not used – set to 0
Bit 6	Not used – set to 0
Bit 7	Not used – set to 0

If coin mech error events are set to ccTalk format, then event Coin Mech Error 0xB7 is given with 1 byte ccTalk coin mech error reason directly from coin mech ccTalk event queue (listed below, from ccTalk Specification v4.6). Otherwise only error events Coin Mech Jam 0xC4 and Coin Mech Return 0xC5 are given.

Code	Error	Code	Error
0	Null event (error)	21	DCE opto timeout
1	Reject coin	22	DCE opto seen
2	Inhibited coin	23	Credit sensor reached too early
3	Multiple window	24	Reject coin (repeated sequential trip)
4	Wake-up timeout	25	Reject slug
5	Validation timeout	26	Reject sensor blocked
6	Credit sensor timeout	27	Games overload
7	Sorter opto timeout	28	Max. coin meter pulses exceeded
8	2nd close coin error	29	Accept gate open closed
9	Accept gate ready	30	Accept gate closed open
10	Credit sensor ready	31	Manifold opto timeout
11	Sorter ready	32	Manifold opto blocked
12	Reject coin cleared	33	Manifold ready
13	Validation sensor ready	34	Security status changed
14	Credit sensor blocked	35	Motor exception
15	Sorter opto blocked	128	Inhibited coin (Type 1)
16	Credit sequence error	...	Inhibited coin (Type n)
17	Coin going backwards	159	Inhibited coin (Type 32)
18	Coin too fast (over credit sensor)	253	Data block request
19	Coin too slow (over credit sensor)	254	Coin return mechanism activated (flight deck open)
20	C.O.S. (coin-on-string) mechanism activated	255	Unspecified alarm code

Copyright Innovative Technology Ltd 2013

10 UPDATING DEVICES IN SSP

As part of our continued development and improvement, Innovative Technology Ltd periodically releases new dataset or firmware for our validators. This could be for improved acceptance, additional features or security updates.

We recommend that network connected cabinets and applications communicating in SSP have the functionality to update the devices attached through the application software. We can provide DLLs and libraries to assist with this development. Please contact your local support office with your requirements for more assistance.

This section outlines the software processes involved in updating a validator with a new dataset/firmware file. Implementation of this process allows a validator to be updated from a remote location using the host machine software.

NOTE
This is a complex operation and failure to implement correctly may damage units. Please exercise extreme caution when writing firmware to the device.

10.1 FILE STRUCTURE

A firmware/dataset file is composed of the following sections:

EUR02604_NV02004141498000_IF_01.bv1

Header block (128 bytes)	**RAM block** (variable length ≈10kb)	**Firmware/Dataset block** (variable length ≈ 300kb)

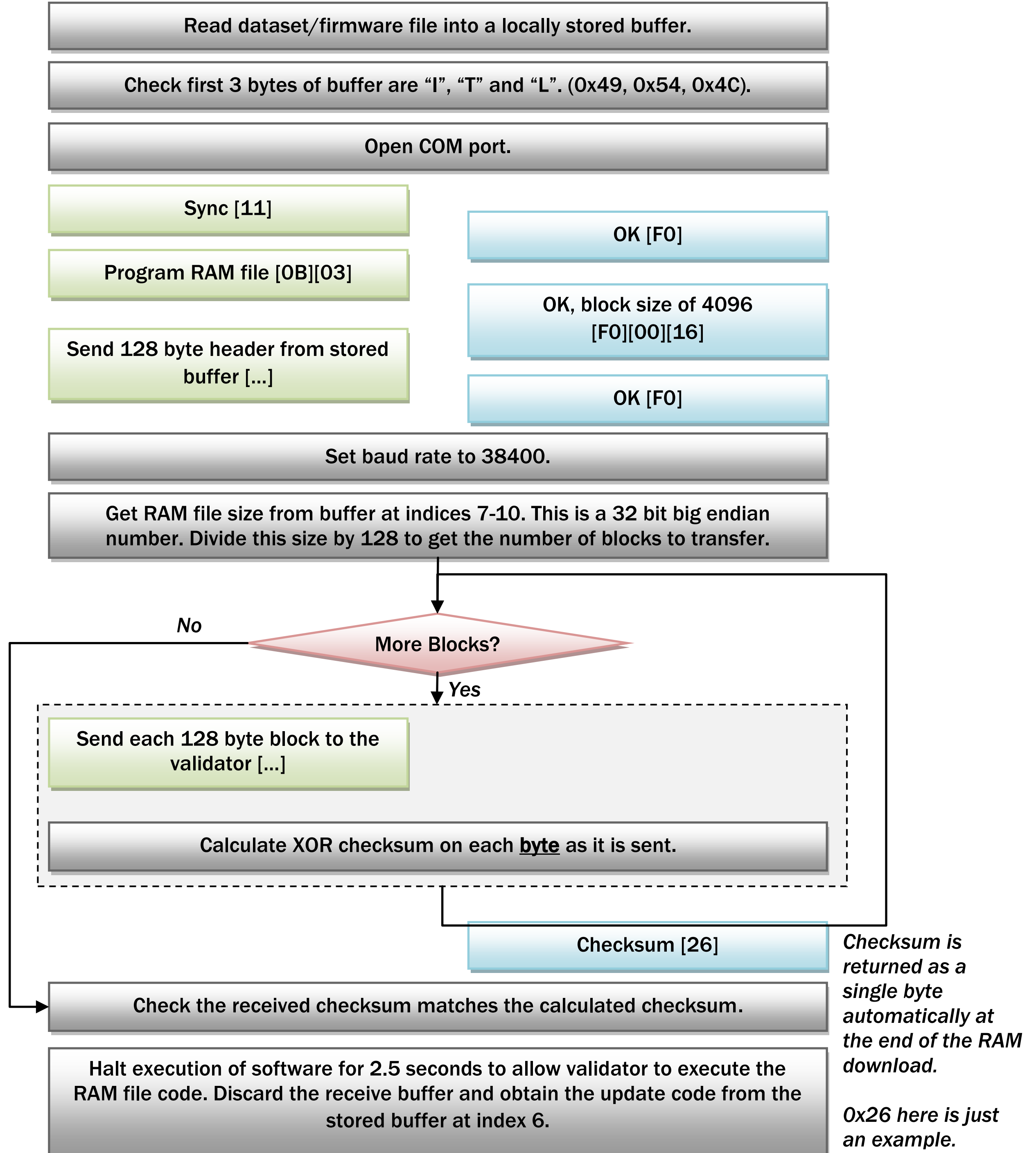
The **Header block** contains details about the file including validator type, versioning information and download configuration data used by the validator during the update.

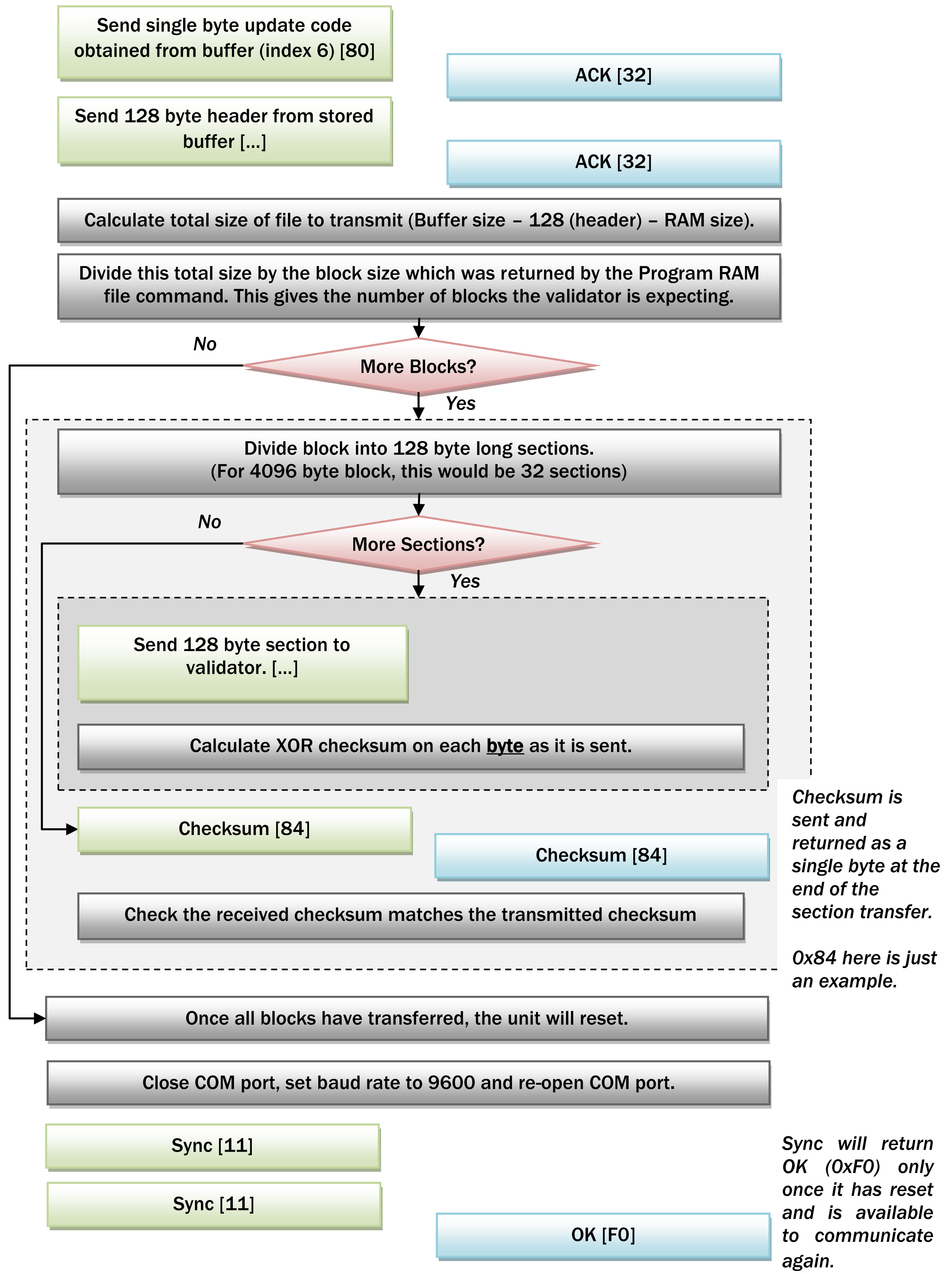
The **RAM block** contains the code run on the validator during the update process. This controls the operation of the validator during the update process and is erased on reset or power loss.

The **Firmware/Dataset block** contains the update that will be applied to the validator.

10.2 PROCESS OVERVIEW

Below is a summary of the download process, more detail on each stage of the download is included in later sections.





Checksum is sent and returned as a single byte at the end of the section transfer.

0x84 here is just an example.

Sync will return OK (0xF0) only once it has reset and is available to communicate again.

10.3 PROCESS DETAILS

10.3.1 CONFIRMING THE PRESENCE OF THE VALIDATOR

This is done by sending the SSP Sync command (0x11) to the validator.

The response received should be OK (0xF0) confirming the validator is responding to comms.

```
cmd.CommandData[0] = 0x11;
cmd.CommandDataLength = 0x01;
if (!SSPSendCommand(cmd, info) || cmd.ResponseData[0] != 0xF0)
       return false;
```

10.3.2 SENDING THE PROGRAMMING COMMAND AND RETRIEVING THE BLOCK SIZE

The next SSP command is sent to prepare the validator to begin the programming procedure.

The command sent is 0x0B, this is a two byte command, the second byte contains the programming the validator can expect to receive. In this case the second byte should be 0x03, this is the RAM programming command. The RAM file is transferred to the validator before the firmware/dataset file is transferred. The validator updates itself based on this RAM code rather than on code stored previously in the validator.

The OK (0xF0) response should be received from this command along with the size of the block that the validator will expect data to be transferred in. This size is a 2 byte little endian number and follows the OK response.

```
cmd.CommandData[0] = 0x0B;
cmd.CommandData[1] = 0x03;
cmd.CommandDataLength = 0x02;
if (!SSPSendCommand(cmd, info))
       return false;

// Obtain block size from response (16 bit / 2 bytes)
if (cmd.ResponseData[0] = 0xF0)
{
       short blockSize;
       blockSize = cmd.ResponseData[1];
       blockSize += (short)(cmd.ResponseData[2] << 8);
}
```

10.3.3 SENDING THE HEADER BLOCK TO THE VALIDATOR

The header block is simply the first 128 bytes of the main firmware/dataset file. This is transmitted to the validator as an SSP command.

The command data simply contains the first 128 bytes from the file with a length set to 128.

The response received should be OK (0xF0) if the file is the correct type for this validator, or HEADER_FAIL (0xF9) if this file cannot be used to update this validator.

```
// Send header file
for (int i = 0; i < 128; ++i)
       cmd.CommandData[i] = (byte)fileBuffer[i];
cmd.CommandDataLength = 128;

if (SSPSendCommand(cmd, info))
{
       if (cmd.ResponseData[0] == 0xF9)
       {
              return false;
       }
}
```

10.3.4 SENDING THE RAM FILE TO THE VALIDATOR

From this point, the SSP command packet format is not used. The data is written directly to the validator using the serial port.

The RAM file is stored in the main firmware/dataset file after the header block. Its size is retrieved from the file. The size is a 32 bit big endian number stored in the indices 7 to 10 of the main file.

```
int totalRamSize = 0;
totalRamSize += fileBuffer[7] << 24;
totalRamSize += fileBuffer[8] << 16;
totalRamSize += fileBuffer[9] << 8;
totalRamSize += fileBuffer[10];
```

Once the size has been obtained then the RAM file can be sent to the validator, this is done by writing 128 byte blocks to the validator until the whole file has been transmitted. If the RAM file does not fit equally into 128 byte blocks then the remainder of the file should be transmitted in a block of the leftover size.

```
short numRamBlocks = (short)(totalRamSize / 128);
byte[] blockToWrite = new byte[128];
byte checksum = 0;
for (int currentRamBlock = 0; currentRamBlock < numRamBlocks; ++currentRamBlock)
{
        for (int j = 0; j < 128; ++j)
        {
                blockToWrite[j] = (byte)fileBuffer[128 + (currentRamBlock * 128) + j];
                checkSum ^= blockToWrite[j];
        }
        comPort.Write(blockToWrite, 0, 128);

}
byte remainder = (byte)(totalRamSize % 128);
if (remainder != 0)
{
        for (int i = 0; i < remainder; ++i)
        {
                blockToWrite[i] = (byte) fileBuffer[128 + (currentRamBlock * 128) + i];
                checkSum ^= blockToWrite[i];
        }
        comPort.Write(blockToWrite, 0, remainder);
}
```

An XOR checksum should be kept on each transferred byte, once the whole RAM file has been transferred the validator will send a response. The first byte of this response will contain the checksum of the RAM file as calculated by the validator, this should match the one calculated as the file was transferred. If they do not match then the download process should be aborted and restarted.

```
// Wait for response
stopWatch.Restart();
while (!downloadResponse)
{
        // Wait one second then assume timeout
        if (stopWatch.ElapsedMilliseconds > 1000)
        {
                downloading = false;
                return false;
        }
}
downloadResponse = false;

// Check that the checksum matches
if (rxData[0] != checkSum)
        return false;
```

After the RAM file has been successfully transmitted to the validator, the execution of the program should halt for 2.5 seconds to allow the validator to run the RAM code.

10.3.5 SENDING THE FIRMWARE/DATASET DATA TO THE VALIDATOR

The main data containing the firmware/dataset can now be transmitted to the validator. This is done by following these steps:

- Clear the receive buffer on the COM port of any old data.
- Obtain the update code from the firmware/dataset file, this is a single byte located at index 6.

- Write this update code directly to the validator as a single byte.

```
// Get update code from byte index 6 of the file
blockToWrite[0] = fileBuffer[6];

// Write the single byte
comPort.Write(blockToWrite, 0, 1);
```

- The validator will send a response, this response should be an Acknowledgement byte of 0x32. If this response is not received, abort the download.

```
// Wait for a response
stopWatch.Restart();
while (!downloadResponse)
{
        // Wait one second then assume timeout
        if (stopWatch.ElapsedMilliseconds > 1000)
        {
                downloading = false;
                return false;
        }
}
downloadResponse = false;

// If no ACK, abort
if (rxData[0] != 0x32)
        return false;
```

- Send the header block again, directly write the first 128 bytes of the firmware/dataset file to the validator as a single block.
- Wait for an ACK response of 0x32. If this isn't received, abort the download.

```
// Resend header
comPort.Write(fileBuffer, 0, 128);

// Wait for a response
stopWatch.Restart();
while (!downloadResponse)
{
        if (stopWatch.ElapsedMilliseconds > 1000)
        {
                downloading = false;
                return false;
        }
}
downloadResponse = false;

// If no ACK, abort
if (rxData[0] != 0x32)
        return false;
```

- Obtain the total size of the download, this can be calculated by obtaining the whole file size from the buffer, then subtracting 128 to cover the header block, then subtracting the RAM file size as calculated previously. This will leave the total size of the data to be transferred.

Copyright Innovative Technology Ltd 2013

- Divide this total size by the block size obtained with the programming command in section 3.1.2. This gives the total number of blocks to transfer to the validator.

```
int totalDownloadSize = fileBuffer.Length - totalRamSize - 128;
short totalDownloadBlocks = (short)(totalDownloadSize / blockSize);
```

- For each block, break the block size into segments of 128 bytes and write these to the validator. Keep an XOR checksum on each byte written.
- At the end of a complete block, send the calculated checksum to the validator as a single byte.
- The validator will send a response where the first byte contains the checksum as calculated by the validator. If this checksum matches the one calculated as the blocks are sent then this block was a success and the next can be sent in the same way.

```
// For each block in the total to transfer
for (currentDownloadBlock = 0; currentDownloadBlock < totalDownloadBlocks; ++currentDownloadBlock)
{
        // Reset checksum for each block
        checkSum = 0;

        // Copy the bytes from each block into an array to send, only send 128 bytes at a time
        int breakdown = blockSize / 128;
        for (int i = 0; i < breakdown; ++i)
        {
                for (int j = 0; j < 128; ++j)
                {
                        // Skip the header (128 bytes) + the RAM file +
                        // the already written blocks + the broken down data sent so far
                        blockToWrite[j] = (byte)fileData[128 + totalRamSize +
                (currentDownloadBlock * blockSize) + j + (i*128)];

                        checkSum ^= blockToWrite[j]; // Keep track of checksum for this block
                }

                // Write this 128 byte block to the validator
                comPort.Write(blockToWrite, 0, 128);
        }

        // Write the checksum to the validator
        blockToWrite[0] = checkSum;
        comPort.Write(blockToWrite, 0, 1);

        // Wait for a response
        stopWatch.Restart();
        while (!downloadResponse)
        {
                if (stopWatch.ElapsedMilliseconds > 1000)
                {
                        downloading = false;
                        return false;
                }
        }
        downloadResponse = false;

        // Response received, does checksum match?
        if (rxData[0] != checkSum)
        {
                downloading = false;
                return false;
        }

// This block transferred successfully and we can write the next
}
```

- In the same way as the RAM file, if the block size does not exactly fit into the total size then the remainder should be sent in divisions of 128 bytes or less until all bytes have been transferred.
- Once all bytes have been transferred the validator will reset.

10.3.6 CHECKING THE SUCCESS OF THE TRANSFER

From here the mode of sending data should be restored to the SSP packet format.

The COM port should be closed, the baud rate set back to 9600 and then re-opened.

An SSP Poll command should be sent to the validator repeatedly until it responds with an OK (0xF0) response.

```
// Reset baud to 9600
comPort.Close();
comPort.BaudRate = 9600;
comPort.Open();

// Send sync to determine if validator back online
bool online = false;
stopWatch.Restart();
do
{
	cmd.CommandData[0] = 0x11;
	cmd.CommandDataLength = 0x01;
	online = SSPSendCommand(cmd, info);
	if (stopWatch.ElapsedMilliseconds > 10000)
	{
		return false;
	}
} while (!online);

return true;
```

At this point the firmware/dataset has been successfully transferred and the validator is ready for use.

11 LIBRARY REFERENCE

This section describes the libraries provided by Innovative Technology LTD. to assist developers to implement eSSP.

Two ITL libraries are currently supported in the form of Microsoft Windows DLLs.
ITLLib.dll is provided for /Net platform development (C# and VB.Net).
ITLSSPProc.dll is provided for C++ and VB6 development.

11.1 ITLLIB.DLL

The ITLLib.dll is a dynamic link library designed to be used with Windows .NET applications to help with the design of software used to communicate in either SSP or eSSP with an ITL validator.

The library contains useful functions to simplify packet construction, encryption, and writing bytes to and retrieving bytes from a com port.

This section provides a guide to what is contained inside this library and how it can help a user to implement SSP or eSSP communication.

11.1.1 LIBRARY STRUCTURE

The ITLLib.dll contains a number of public classes available for instantiation. All except one of these public classes are simply collections of variables used by the library and instantiated and filled with data by the developer.

These classes consist of:

```
public class SSP_KEYS
    {
        public UInt64 Generator;
        public UInt64 Modulus;
        public UInt64 HostInter;
        public UInt64 HostRandom;
        public UInt64 SlaveInterKey;
        public UInt64 SlaveRandom;
        public UInt64 KeyHost;
        public UInt64 KeySlave;
    };
```

```
public class SSP_FULL_KEY
    {
        public UInt64 FixedKey;
        public UInt64 VariableKey;
    };
```

```
public class SSP_COMMAND
    {
        public SSP_FULL_KEY Key = new SSP_FULL_KEY();
        public Int32 BaudRate = 9600;
        public UInt32 Timeout = 500;
        public string ComPort;
        public byte SSPAddress = 0;
        public byte RetryLevel = 3;
        public bool EncryptionStatus = false;
        public byte CommandDataLength;
        public byte[] CommandData = new byte[255];
        public PORT_STATUS ResponseStatus = new PORT_STATUS();
        public byte ResponseDataLength;
        public byte[] ResponseData = new byte[255];
        public UInt32 encPktCount;
        public byte sspSeq;
    };
```

```
public class SSP_COMMAND_INFO
    {
        public bool Encrypted;
        public SSP_PACKET Transmit = new SSP_PACKET();
        public SSP_PACKET Receive = new SSP_PACKET();
        public SSP_PACKET PreEncryptedTransmit = new SSP_PACKET();
        public SSP_PACKET PreEncryptedRecieve = new SSP_PACKET();
    };
```

The other public class available in this library contains methods that can be used by the developer, it is named SSPComms and is covered in detail in the following section.

11.1.2 SSPCOMMS PUBLIC METHODS

OPENSSPCOMPORT (SSP_COMMAND)

This method uses the Windows class System.IO.Ports.SerialPort to open a com port. The port that is opened along with the baud rate is determined from an instance of the SSP_COMMAND class that is passed as the only parameter. This function also adds an event handler to the port for received data. The function returns a boolean value, true indicates the port is open, false indicates that the port was not opened.

Example

```
SSP_COMMAND sspc = new SSP_COMMAND();
sspc.ComPort = "COM12";
sspc.BaudRate = 9600;
if (sspLib.OpenSSPComPort(sspc))
{
        // Port opened successfully
}
```

CLOSECOMPORT ( )

This method simply closes the com port the developer opened with the OpenSSPComPort method. It takes no parameters and returns a boolean value, true indicates the port was closed, false indicates an exception was throw when trying to close the port.

Example

```
if (sspLib.CloseComPort())
{
        // Port was closed
}
```

SSPSENDCOMMAND (SSP_COMMAND, SSP_COMMAND_INFO)

This method involves a number of steps in order to receive a packet's data, compile the packet, optionally encrypt it and then transmit it to the validator. The method requires an SSP_COMMAND instance to be passed as a parameter. As long as the correct data has been entered into the SSP_COMMAND instance then this method will transmit a complete packet to the validator.

If the SSP_COMMAND variable EncryptionStatus is set to true, a key must have been negotiated with the validator before a packet can be encrypted and sent.

The SSP_COMMAND_INFO instance which is passed as the other parameter to this method is for the use of the developer, it will be filled with information about the transmitted and received packets.

Copyright Innovative Technology Ltd 2013

This method will return true under the following conditions:

- A packet has been sent successfully to the validator and a response received.
- An encrypted packet has been sent successfully to the validator and an encrypted response received.

This method will return false under the following conditions:

- There was a problem with the compilation of a packet, either encrypted or not.
- There was a problem writing to the port.
- A response was not received from the validator within the timeout period.
- There was a problem with the checksum calculation.
- The count of the slave and the host are mismatched.

In addition to returning false, further information can be obtained about the specific reason for the failure to send a packet. The library will set a variable inside the SSP_COMMAND instance named ResponseStatus. This variable is of the following enumeration:

```
public enum PORT_STATUS
{
        PORT_CLOSED,
        PORT_OPEN,
        PORT_ERROR,
        SSP_REPLY_OK,
        SSP_PACKET_ERROR,
        SSP_CMD_TIMEOUT,
};
```

<u>Example</u>

```
sspc.CommandData[0] = 0x11;
sspc.CommandDataLength = 1;
if (sspLib.SSPSendCommand(sspc, sspi))
{
        // Command was sent successfully
}
```

INITIATESSPHOSTKEYS (SSP_KEYS, SSP_COMMAND)

This method is involved with the encryption process used in eSSP. It sets up an instance of the SSP_KEYS structure to begin the encryption process, this involves generating the random 64 bit prime numbers required and creating the host intermediate key to send to the slave.

<u>Example</u>

```
SSP_KEYS sspk = new SSP_KEYS();
sspLib.InitiateSSPHostKeys(sspk, sspc);
```

Copyright Innovative Technology Ltd 2013

CREATESSPHOSTENCRYPTIONKEY (SSP_KEYS)

This method is the final public method that needs to be used in the library. It takes a SSP_KEYS instance as a parameter and uses the information contained within to generate the final encryption key. This key can then be set in the SSP_COMMAND instance, this allows the SSPSendCommand method to encrypt packets and send them if the variable EncryptionStatus in the SSP_COMMAND instance is set to true.

Example

```
sspLib.CreateSSPHostEncryptionKey(sspk);
sspc.Key.FixedKey = 0x12345671234567;
sspc.Key.VariableKey = sspk.KeyHost;
```

11.2 ITLSSPPROC.DLL

This section describes the interface to the ITLSSPProc eSSP DLL.

This DLL has been developed to assist in the implementation of SSP and the encryption required for eSSP in Windows based system. It provides a mechanism to format and send packets to an SSP target across a serial link from a Windows host. It also provides the routines required for setting keys and encrypting packets.

11.2.1 LIBRARY STRUCTURE

The DLL interface requires defined structures:

C STRUCTURE DEFINITIONS.

This structure is used by the host to store the full encryption key The FixedKey bytes are defined by the host and must match the slave fixed key.

```
typedef struct{
	unsigned __int64 FixedKey;		// 8 byte number for fixed host key
	unsigned __int64 EncryptKey;		// 8 Byte number for variable key
}SSP_FULL_KEY;
```

This structure is required for the key exchange process.

```
typedef struct{
unsigned __int64 Generator;
	unsigned __int64 Modulus;
unsigned __int64 HostInter;
	unsigned __int64 HostRandom;
unsigned __int64 SlaveInterKey;
	unsigned __int64 SlaveRandom;
unsigned __int64 KeyHost;
	unsigned __int64 KeySlave;
}SSP_KEYS;
```

Port status code enumeration for ResponseStatus element.

```
typedef enum{
	PORT_CLOSED,
	PORT_OPEN,
	PORT_ERROR,
	SSP_REPLY_OK,
	SSP_PACKET_ERROR,
	SSP_CMD_TIMEOUT,
}PORT_STATUS;
```

Structure to define an SSP command.

```
typedef struct{
       SSP_FULL_KEY Key;     // the full key
       unsigned long BaudRate;  // baud rate of the packet
       unsigned long Timeout;  // how long in ms to wait for reply from slave
       unsigned char PortNumber;  // the serial com port number of the host
       unsigned char SSPAddress; // the SSP address of the slave
       unsigned char RetryLevel;   // how many retries to slave for non-response
       unsigned char EncryptionStatus;   // encrypted command 0 – No, 1 - Yes
       unsigned char CommandDataLength; // Number of bytes in the command
       unsigned char CommandData[255];  //  Array containing the command bytes
       unsigned char ResponseStatus;// Response Status (PORT_STATUS enum)
       unsigned char ResponseDataLength;   // how many bytes in the response
       unsigned char ResponseData[255]; // an array of response data
       unsigned char IgnoreError; // suppress error box (0 – display,1- suppress)
}SSP_COMMAND;
```

Structure to define an SSP Packet.

```
typedef struct{
       unsigned short packetTime;    // the time in ms taken for reply response
       unsigned char PacketLength;  // The length of SSP packet
       unsigned char PacketData[255]; // packet data array
}SSP_PACKET;
```

Structure to define SSP packet info for log file and display purposes.

```
typedef struct{
       unsigned char* CommandName;
       unsigned char* LogFileName;
       unsigned char Encrypted;
       SSP_PACKET Transmit;
       SSP_PACKET Receive;
       SSP_PACKET PreEncryptedTransmit;
       SSP_PACKET PreEncryptedRecieve;
}SSP_COMMAND_INFO;
```

Structure to hold information about communications ports used on the host when more than one is being used.

```
typedef struct{
       unsigned char NumberOfPorts;
       unsigned char PortID[MAX_PORT_ID];
       unsigned long BaudRate[MAX_PORT_ID];
}PORT_CONFIG;
```

Copyright Innovative Technology Ltd 2013

VISUAL BASIC STRUCTURE DEFINITIONS.

```
Public Type EightByteNumber
    LoValue As Long
    Hivalue As Long
End Type
```

```
Public Type SSP_FULL_KEY
    FixedKeyLowValue As Long
    FixedKeyHighValue As Long
    EncryptKeyLowValue As Long
    EncryptkeyHighValue As Long
End Type
```

```
Public Type SSP_KEYS
    Generator As EightByteNumber
    Modulus As EightByteNumber
    HostInter As EightByteNumber
    HostRandom As EightByteNumber
    SlaveInterKey As EightByteNumber
    SlaveRandom As EightByteNumber
    KeyHost As EightByteNumber
    KeySlave As EightByteNumber
End Type
```

```
Public Enum PORT_STATUS
    PORT_CLOSED
    port_open
    PORT_ERROR
    ssp_reply_ok
    SSP_PACKET_ERROR
    SSP_CMD_TIMEOUT
End Enum
```

```
Public Type SSP_COMMAND
    Key As SSP_FULL_KEY
    BaudRate As Long
    Timeout As Long
    PortNumber As Byte
    sspAddress As Byte
    RetryLevel As Byte
    EncryptionStatus As Byte
    CommandDataLength As Byte
    CommandData(254) As Byte
    ResponseStatus As Byte
    ResponseDataLength  As Byte
    ResponseData(254) As Byte
End Type
```

```
Public Type SSP_PACKET
    PacketTime As Integer
    PacketLength As Byte
    PacketData(254) As Byte
End Type
```

```
Public Type SSP_COMMAND_INFO
    CommandName As String
    LogFileName As String
    Encrypted As Byte
    Transmit As SSP_PACKET
    Recieve As SSP_PACKET
    PreEncryptTransmit As SSP_PACKET
    PreEncryptRecieve As SSP_PACKET
End Type
```

```
Public Type PORT_CONFIG
       NumberOfPorts As Byte
       PortID(MAX_PORT_ID) As Byte
       BaudRate(MAX_PORT_ID) as Long
End Type
```

11.2.2 API DECLARATIONS

VISUAL BASIC™ 6

```
Public Declare Function OpenSSPComPort Lib "ITLSSPProc.dll" _
       (ByRef sspc As SSP_COMMAND) As Integer
Public Declare Function CloseSSPComPort Lib "ITLSSPProc.dll" () As Integer
Public Declare Function OpenSSPComPort2 Lib "ITLSSPProc.dll" _
       (ByRef sspc As SSP_COMMAND) As Integer
Public Declare Function CloseSSPComPort2 Lib "ITLSSPProc.dll" () As Integer
Public Declare Function OpenSSPComPortUSB Lib "ITLSSPProc.dll" _
        (ByRef sspc As SSP_COMMAND) As Integer
Public Declare Function CloseSSPComPortUSB Lib "ITLSSPProc.dll" () As Integer
Public Declare Function OpenSSPMulipleComPorts Lib “ITLSSPProc.dll” _
        (ByRef pt As PORT_CONFIG) As Integer
Public Declare Function CloseSSPMultiplePorts Lib “ITLSSPProc.dll” () As Integer
Public Declare Function InitiateSSPHostKeys Lib "ITLSSPProc.dll" _
        (ByRef Key As SSP_KEYS, ByRef sspc As SSP_COMMAND) As Integer
Public Declare Function CreateSSPHostEncryptionKey Lib "ITLSSPProc.dll" _
        (ByRef Key As SSP_KEYS) As Integer
Public Declare Function SSPSendCommand Lib "ITLSSPProc.dll" _
        (ByRef sspc As SSP_COMMAND, ByRef sspinfo As SSP_COMMAND_INFO) As Integer
```

11.2.3 FUNCTION DECLARATIONS.

OPENSSPCOMPORT

Parameters:

Pointer to SSP_COMMAND structure

Returns:

WORD 0 for fail, 1 for success

Description:

Opens a serial communication port for SSP data transmission and reception on the host.

Requirements before calling:

SSP_COMMAND structure elements BaudRate and PortNumber need to be correctly filled.

Result after calling:

If function returns 1, host serial port PortNumber is now open for serial comms.

OPENSSPCOMPORT2

Parameters:

Pointer to SSP_COMMAND structure

Returns:

WORD 0 for fail, 1 for success

Description:

Opens a serial communication port for SSP data transmission and reception on the host. This opens an additional com port to the port in OpenSSPComPort so that two devices with different serial ports may be used from the same host.

Requirements before calling:

SSP_COMMAND structure elements BaudRate and PortNumber need to be correctly filled. One of the SSP devices used when two ports are open must have an SSP address of 0. (SMART payout or BNV).

Result after calling:

If function returns 1, host serial port PortNumber is now open for serial comms.

OPENSSPCOMPORTUSB

Parameters:

Pointer to SSP_COMMAND structure

Returns:

WORD 0 for fail, 1 for success

Description:

Opens a serial communication port for SSP data transmission and reception on the host. This function is used when the host has two or more SSP devices (with different SSP address) connected to the same SSP bus.

Requirements before calling:

SSP_COMMAND structure elements BaudRate and PortNumber need to be correctly filled.

Result after calling:

If function returns 1, host serial port PortNumber is now open for serial comms.

OPENSSPMULIPLECOMPORTS

Parameters:

Pointer to PORT_CONFIG structure.

Returns:

WORD 0 for fail, 1 for success.

Description:

Opens multiple serial communication ports for SSP data transmission and reception on the host. The details of which ports to open are contained within the PORT_CONFIG structure passed as a parameter to this function. This function is used when multiple devices are connected to different ports on the host machine.

Requirements before calling:

PORT_CONFIG structure elements NumberOfPorts, PortID and BaudRate need to be correctly filled.

Result after calling:

If function returns 1, multiple host serial ports specified in the PORT_CONFIG structure are now open.

CLOSESSPCOMPORT

Parameters:

None

Returns:

WORD 0 for fail, 1 for success

Description:

Closes the serial communication port on the host corresponding to the OpenSSPComPort function

Requirements before calling:

An open communication port with PortNumber opened in OpenSSPComPort. Note that calling this function if the port is already closed will have no effect and will still return 1.

Result after calling:

If function returns 1, host serial port PortNumber is now closed for serial comms.

CLOSESSPCOMPORT2

Parameters:

None

Returns:

WORD 0 for fail, 1 for success

Description:

Closes the serial communication port on the host corresponding to the OpenSSPComPort2 function

Requirements before calling:

An open communication port with PortNumber opened in OpenSSPComPort2. Note that calling this function if the port is already closed will have no effect and will still return 1.

Result after calling:

If function returns 1, host serial port PortNumber is now closed for serial comms.

CLOSESSPCOMPORTUSB

Parameters:

None

Returns:

WORD 0 for fail, 1 for success

Description:

Closes the serial communication port on the host corresponding to the OpenSSPComPortUSB function

Requirements before calling:

An open communication port with PortNumber opened in OpenSSPComPortUSB. Note that calling this function if the port is already closed will have no effect and will still return 1.

Result after calling:

If function returns 1, host serial port PortNumber is now closed for serial comms.

CLOSESSPMULTIPLEPORTS

Parameters:

None

Returns:

WORD 0 for fail, 1 for success

Description:

Closes all the serial ports on the host that were opened with the OpenSSPMulipleComPorts function.

Requirements before calling:

Previously opened communications ports that were initially opened using the OpenSSPMulipleComPorts function.

Result after calling:

If function returns 1 then all the ports opened with OpenSSPMulipleComPorts are now closed.

INITIATESSPHOSTKEYS

Parameters:

Pointer to the start of SSP_KEY structure,
Pointer to SSP_COMMAND structure

Returns:

WORD 0 for fail, 1 for success

Description:

Function to create encryption Modulus, Generator and Host Inter numbers. These numbers are sent to the slave during the key exchange process.

Requirements before calling:

SSP_COMMAND structure element PortNumber needs to be correctly filled with the host serial port number.

Result after calling:

SSP encryption packet counter is reset to 0 for that host port number.
SSP_KEY structure will be filled with number values in array order:

Generator	valid
Modulus	valid
HostInter	valid
HostRandom	empty
SlaveInterKey	empty
SlaveRandom	empty
KeyHost	empty
KeySlave	empty

CREATESSPHOSTENCRYPTIONKEY

Parameters:

Pointer to the start of SSP_KEY structure,

Returns:

WORD 0 for fail, 1 for success

Description:

Call this function to create your host key using the SSP_KEY structure populated first by the InitiateSSPHostKeys function. This host key will then match the slave key.

Requirements before calling:

An SSP_KEY structure populated by call InitiateSSPHostKeys, then sending the Generator and Modulus numbers to the slave (via SSP packets) to populate the SlaveInterKey element of this structure.

Result after calling:

The KeyHost element of the SSP_KEYS structure contains the 64 bit encryption key to combine with the 64 bit fixed key of the host to create the full 128-bit eSSP encryption key for this system.

SSPSENDCOMMAND

Parameters:

Pointer to SSP_COMMAND structure.
Pointer to SSP_COMMAND_INFO structure.

Returns:

WORD 0 for fail, 1 for success

Description:

Compiles a full ssp packet given a command array, with optional SSP encryption and sends to the slave. The host then waits for a reply, checks its validity and decrypts if required. The function will retry for the number of times specified in RetryLevel parameter after waiting Timeout milliseconds for a response from the slave.

Requirements before calling:

An open communication port with PortNumber opened in one of the OpenSSPComPort functions.

Result after calling:

The function returns 1 for a successful transaction – the SSP_COMMAND structure elements ResponseData and ResponseDataLength contains the slave reply data and the ResponseStatus element will be set to SSP_REPLY_OK.

If the function returns 0, the SSP_COMMAND structure elements ResponseData and ResponseDataLength will contain invalid data and the ResponseStatus element will contain the reason for failure as one of the PORT_STATUS enumeration elements.

12 APPENDIX

A – LAST REJECT CODES

Code (Hex)	Reject Reason
0x00	Note accepted
0x01	Note length incorrect
0x02	Invalid note
0x03	Invalid note
0x04	Invalid note
0x05	Invalid note
0x06	Channel inhibited
0x07	Second note inserted
0x08	Host rejected note
0x09	Invalid note
0x0A	Invalid note read
0x0B	Note too long
0x0C	Validator disabled
0x0D	Mechanism slow/stalled
0x0E	Strimming attempt
0x0F	Fraud channel reject
0x10	No notes inserted
0x11	Peak detect fail
0x12	Twisted note detected
0x13	Escrow time-out
0x14	Bar code scan fail
0x15	Invalid note read
0x16	Invalid note read
0x17	Invalid note read
0x18	Invalid note read
0x19	Incorrect note width
0x1A	Note too short

B – LANGUAGE GUIDES

B.1 – SENDING A COMMAND

C#

```
commandStructure.CommandData[0] = 0x11;
commandStructure.CommandDataLength = 0x01;
sspLibrary.SSPSendCommand(commandStructure, infoStructure);
```

C++ Windows

```
CommandStructure->CommandData[0] = (char)0x11;
CommandStructure->CommandDataLength = (char)0x01;
sspLibrary->SSPSendCommand(commandStructure, infoStructure);
```

Visual Basic

```
commandStructure.CommandDataLength = 1
commandStructure.CommandData(0) = &H11
sspLibrary.SSPSendCommand(commandStructure, commandInfo)
```

B.2 – RECEIVING A RESPONSE

C#

```
if (commandStructure.ResponseData[0] == 0xF0)
{
// Unit successfully received command and is acting on it
} else {
// There was a problem with sending the command, or carrying out the
// command
}
```

C++ Windows

```
if (commandStructure->ResponseData[0] == (char)0xF0)
{
// Unit successfully received command and is acting on it
} else {
// There was a problem with sending the command, or carrying out the
// command
}
```

Visual Basic

```
If (commandStructure.ResponseData(0) = &HF0) Then
‘ Unit successfully received command and is acting on it
Else
 ‘ There was a problem with sending the command, or carrying out the
 ‘ command
End If
```

C – KEY NEGOTIATION

C++ Example

```
// assuming that the com port is open and the command structure has been initialised

// make sure encryption is off
commandStructure->EncryptionStatus = false;

// send sync
commandStructure->CommandData[0] = 0x11;
commandStructure->CommandDataLength = 0x01;
SSPSendCommand(commandStructure, infoStructure);

// generate the random prime numbers for the Generator and Modulus
InitiateSSPHostKeys(keys, commandStructure);

// send generator
commandStructure->CommandData[0] = 0x4A;
commandStructure->CommandDataLength = 9;
for (int i = 0; i < 8; i++)
{
    commandStructure->CommandData[i + 1] = (char)(keys->Generator >> (8 * i));
}
SSPSendCommand(commandStructure, infoStructure);

// send modulus
commandStructure ->CommandData[0] = 0x4B;
commandStructure ->CommandDataLength = 9;
for (int i = 0; i < 8; i++)
{
    commandStructure->CommandData[i + 1] = (char)(keys->Modulus >> (8 * i));
}
SSPSendCommand(commandStructure, infoStructure);

// send key exchange
commandStructure ->CommandData[0] = 0x4C;
commandStructure ->CommandDataLength = 9;
for (int i = 0; i < 8; i++)
{
    commandStructure->CommandData[i + 1] = (char)(keys->HostInter >> (8 * i));
}
SSPSendCommand(commandStructure, infoStructure);

keys->SlaveInterKey = 0;
for (int i = 0; i < 8; i++)
{
    keys->SlaveInterKey += (ULONG) commandStructure->ResponseData[1 + i] << (8 * i);
}

// generate key
CreateSSPHostEncryptionKey(keys);

// set full encryption key in command structure
commandStructure->Key.FixedKey = 0x0123456701234567;
commandStructure->Key.EncryptKey = keys->KeyHost;

cmd->EncryptionStatus = true; // turn on encrypting from this point
```

C# Example

```
// assuming that the com port is open and the command structure has been initialised

// send sync
commandStructure.CommandData[0] = 0x11;
commandStructure.CommandDataLength = 0x01;
SSPSendCommand(commandStructure, infoStructure);

// generate the random prime numbers for the Generator and Modulus
eSSP.InitiateSSPHostKeys(keys, commandStructure);

// send generator
commandStructure.CommandData[0] = 0x4A;
commandStructure.CommandDataLength = 9;
for (byte i = 0; i < 8; i++)
{
        commandStructure.CommandData[i + 1] = (byte)(keys.Generator >> (8 * i));
}
SSPSendCommand(commandStructure, infoStructure);

// send modulus
commandStructure.CommandData[0] = 0x4B;
commandStructure.CommandDataLength = 9;
for (byte i = 0; i < 8; i++)
{
        commandStructure.CommandData[i + 1] = (byte)(keys.Modulus >> (8 * i));
}
SSPSendCommand(commandStructure, infoStructure);

// send key exchange
commandStructure.CommandData[0] = 0x4C;
commandStructure.CommandDataLength = 9;
for (byte i = 0; i < 8; i++)
{
        commandStructure.CommandData[i + 1] = (byte)(keys.HostInter >> (8 * i));
}
SSPSendCommand(commandStructure, infoStructure);

keys.SlaveInterKey = 0;
for (byte i = 0; i < 8; i++)
{
        keys.SlaveInterKey += (UInt64)commandStructure.ResponseData[1 + i] << (8 * i);
}

// generate key
eSSP.CreateSSPHostEncryptionKey(keys);

// set full encryption key in command structure
cmd.Key.FixedKey = 0x0123456701234567;
cmd.Key.VariableKey = keys.KeyHost;
```

Visual Basic Example

```
' assuming that the com port is open and the command structure has been initialised
Public Function NegotiateKeyExchange(sspc As SSP_COMMAND, _
        sspInfo As SSP_COMMAND_INFO) As Boolean
Dim sspKey As SSP_KEYS
Dim i As Integer
' DLL call to create Modulus, Generator and Host inter numbers
If InitiateSSPHostKeys(sspKey, sspc) = 0 Then
        MsgBox "Error initiating host key modulus or generator values set to zero", _
        vbExclamation, App.ProductName
        Exit Function
End If
sspc.CommandDataLength = 1
sspc.EncryptionStatus = 0
sspc.CommandData(0) = SYNC_CMD
If Not TransmitSSPCommand(sspc, sspInfo) Then Exit Function
sspc.CommandDataLength = 9
sspc.CommandData(0) = cmd_SSP_SET_GENERATOR
For i = 0 To 3
        sspc.CommandData(1 + i) =
        CByte(RShift(sspKey.Generator.LoValue, 8 * i) And &HFF)
        sspc.CommandData(5 + i) =
        CByte(RShift(sspKey.Generator.Hivalue, 8 * i) And &HFF)
Next I
If Not TransmitSSPCommand(sspc, sspInfo) Then Exit Function
sspc.CommandDataLength = 9
sspc.CommandData(0) = cmd_SSP_SET_MODULUS
For i = 0 To 3
        sspc.CommandData(1 + i) =
        CByte(RShift(sspKey.Modulus.LoValue, 8 * i) And &HFF)
        sspc.CommandData(5 + i) =
        CByte(RShift(sspKey.Modulus.Hivalue, 8 * i) And &HFF)
Next i
If Not TransmitSSPCommand(sspc, sspInfo) Then Exit Function
sspc.CommandDataLength = 9
sspc.CommandData(0) = cmd_SSP_REQ_KEY_EXCHANGE
For i = 0 To 3
        sspc.CommandData(1 + i) =
        CByte(RShift(sspKey.HostInter.LoValue, 8 * i) And &HFF)
        sspc.CommandData(5 + i) =
        CByte(RShift(sspKey.HostInter.Hivalue, 8 * i) And &HFF)
Next i
If Not TransmitSSPCommand(sspc, sspInfo) Then Exit Function
sspKey.SlaveInterKey.LoValue = 0
sspKey.SlaveInterKey.Hivalue = 0
For i = 0 To 3
        sspKey.SlaveInterKey.LoValue = sspKey.SlaveInterKey.LoValue + _
         (CLng(sspc.ResponseData(1 + i)) * (256 ^ i))
        sspKey.SlaveInterKey.Hivalue = sspKey.SlaveInterKey.Hivalue + _
          (CLng(sspc.ResponseData(5 + i)) * (256 ^ i))
Next i
' we can now calculate our host key using the DLL method
If CreateSSPHostEncryptionKey(sspKey) = 0 Then
        MsgBox "Error creating host key", vbExclamation, App.ProductName
        Exit Function
End If
sspc.Key.EncryptKeyLowValue = sspKey.KeyHost.LoValue
sspc.Key.EncryptkeyHighValue = sspKey.KeyHost.Hivalue
NegotiateKeyExchange = True
End Function
```

D – POLL WITH ACK

The poll with ACK command is used in order to avoid missing critical poll events. It works in a very similar way to the poll command except certain specified events will need to be acknowledged before the unit will move on and execute any further note actions. These events will continue to poll until they are acknowledged. Below is a psuedocode example of polling with ACK.

- While polling with ACK.
- If a command requiring acknowledgement is received i.e. Dispensed.
- Perform relevant operations for that command i.e. decrement totals.
- Send event ACK command (0x57) to allow the unit to continue.
- If a command not requiring acknowledgement is received then continue as normal.

E – FIRMWARE VERSIONS AND PROTOCOL LEVEL SUPPORT

NV9USB

Protocol Version	Firmware Version
6	3.27
7	3.33
8	-

NV11

Protocol Version	Firmware Version
6	3.27
7	3.33
8	-

NV200

Protocol Version	Firmware Version
6	4.07
7	4.08
8	4.09

SMART Payout

Protocol Version	Firmware Version
6	4.07
7	4.08
8	4.09

SMART Hopper

Protocol Version	Firmware Version
6	6.03
7	6.09
8	-

Copyright Innovative Technology Ltd 2013

The following tables detail the minimum protocol version for which the associated poll responses will be returned. If the device is set to a protocol level lower than the number detailed below, these responses will not be returned.

This mechanism allows a developer to implement software to control a device to a specific protocol version. The firmware can be updated but no unknown events will be reported back from the poll command until the protocol level is raised.

BANK NOTE VALIDATOR

Event/ State	Protocol Version
Slave Reset	<4
Read Note	<4
Credit Note	<4
Rejecting	<4
Rejected	<4
Stacking	<4
Stacked	<4
Safe Jam	<4
Unsafe Jam	<4
Disabled	<4
Fraud Attempt	<4
Stacker Full	<4
Note cleared from front at reset	4
Note cleared into cash box at reset	4
Cash Box Removed	4
Cash Box Replaced	4
Bar Code Ticket Validated	4
Bar Code Ticket Acknowledge	4
Note path open	6
Channel Disable	7
Initialising (Poll w. ACK response only)	7

Copyright Innovative Technology Ltd 2013

SMART PAYOUT

Event/ State	Protocol Version
Dispensing	4
Dispensed	4
Jammed	4
Halted	4
Floating	4
Floated	4
Time Out	4
Incomplete Payout	4
Incomplete Float	4
Emptying	4
Emptied	4
Payout out of service	6
Note stored in payout	4
Jam Recovery	7
Error During Payout	7
SMART Emptying	4
SMART Emptied	4
Channel Disable	7
Note Transferred to Stacker	8
Note held in bezel	8
Note paid into store at power up	8
Note paid into stacker at power-up	8

NV11

Event/ State	Protocol Version
Dispensing	4
Dispensed	4
Jammed	4
Halted	4
Incomplete Payout	4
Emptying	4
Empty	4
Note stored in payout	4
Note Transferred to Stacker	8
Payout out of service	4
Note paid into stacker at power-up	8
Note paid into store at power up	8
Note dispensed at power up	8
Note Float Removed	4
Note Float Attached	4
Note in Bezel Hold	8
Device Full	4
Channel Disable	7

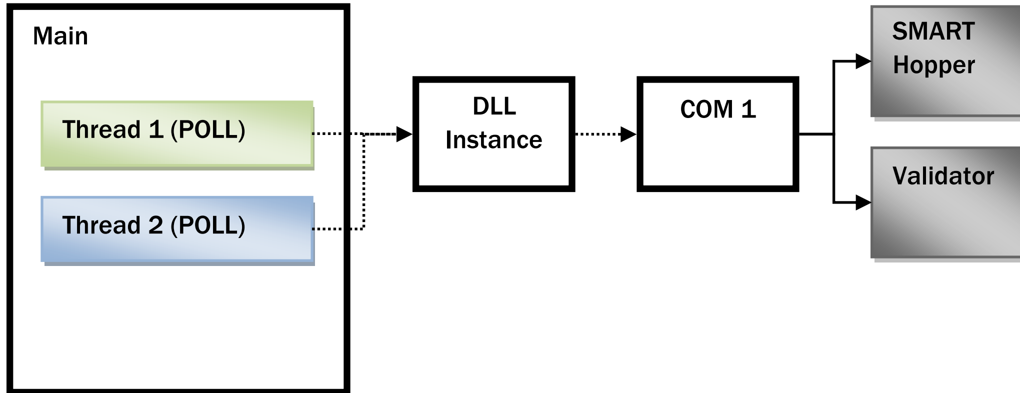
SMART HOPPER

Event/ State	Protocol Version
OK	5
Slave reset	5
Unit disabled	5
Dispensing	5
Dispensed	5
Lid Open	6
Lid Closed	6
Calibration Fail	6
Jammed	5
Halted	5
Floating	5
Floated	5
Time Out	5
Incomplete Payout	5
Incomplete Float	5
Emptying	5
Empty	5
Cash Box Paid	5
Coin Credit	5
Coin mech jammed	5
Coin mech return button pressed	5
Fraud Attempt	5
Low Payout Level	5
SMART Empting	5
SMART Emptied	5

F – SHARING RESOURCES

This section describes firstly the issues with threaded communication and then the best practice. It is intended as a guide.

If the program is setup as shown below, there is a global pointer to the instance of the DLL that both threads use for communication.

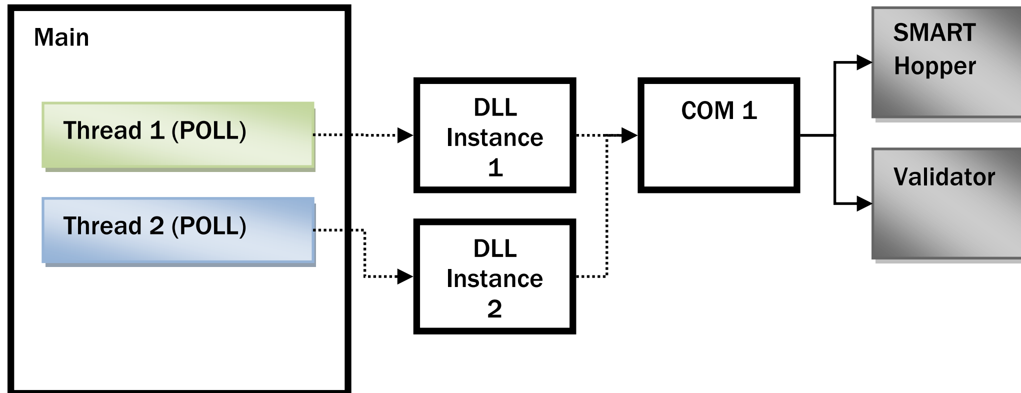


There is only one channel (port) the data can go through to the SMART Hopper and the validator. If SSPSendCommand method is used by both threads without the first call completing and a response being sent back, it will not work.

Time ↓

THREAD 1	THREAD 2
Poll →	
← Ok	
	Set Routing →
	← Ok
Poll →	
	Set Routing →

To get around this, one suggestion has been as follows:



This is not possible as there is an internal flag in the DLL that requires that instance of the DLL to open the port before commands can be sent. This would mean both instances would have to open the port and this is not possible. There is not currently a way to pass/get the open port handle in the DLL instance.

Solution
The suggest way to resolve this is to have the two threads running each with a SendCommand method that checks a global boolean variable called something like LOCK. When a command is being processed, locked is set to true and the SendCommand method waits, polling the LOCKED variable until it is false (a timeout is suggested too to ensue if something goes wrong, the program is not halted).

Time	LOCK	THREAD 1	THREAD 2
↓	1	Poll →	
	1		
	1	← Ok	
	0		
	1		Set Routing →
	1		
	1		← Ok
	0		
	1	Poll →	
	1		
	1		Set Routing Hold
	1	← Ok	Set Routing Hold
	1		Set Routing →

REVISION HISTORY

The commands and responses detailed in this document are based on the following protocol levels and document versions:

Document/Model	Version	Protocol Version	Released
GA138	31	-	

NV9USB	3.39	7	2012-03-20
NV10USB	3.30	7	2011-11-03
BV20	4.07	7	2011-02-25
BV50	4.07	7	2011-10-03
BV100	4.06	6	2010-07-15
NV11	3.39	7	2012-03-20
NV200	4.14	7	2012-01-27
SMART Payout	4.14	8	2012-01-27
SMART Hopper	6.14	8	2012-03-13

INNNOVATIVE TECHNOLOGY LTD			
TITLE	SSP Implementation Guide		
DRAWING NO	AUTHOR	DATE	FORMAT
GA973	SR	2012/05/10	MS Word 2000

ISSUE	RELEASE DATE	MODIFIED BY	COMMENTS
A	2012/03/03	SR	Draft Issue A
B	2012/04/23	SR	Draft Issue B
1	2012/05/03	SR	Release 1
2	2012/05/10	SR	Add chapter 10 – downloading f/w
2.1	2012/05/14	SR	Fix layering issue in initial v2 release (P124) and update example code on P130.
2.2	2013/08/08	AB	