

Milan / Paylink

Application Program Interface

Manual

Aardvark Embedded Solutions Ltd does not accept liability for any errors or omissions contained within this document. Aardvark Embedded Solutions Ltd shall not incur any penalties arising out of the adherence to, interpretation of, or reliance on, this document. Aardvark Embedded Solutions Ltd will provide full support for this product when used as described within this document. Use in applications not covered or outside the scope of this document may not be supported. Aardvark Embedded Solutions Ltd. reserves the right to amend, improve or change the product referred to within this document or the document itself at any time.

Table of Contents

Table of Contents	2
Revision History	6
Introduction	7
Purpose of Document	7
Intended Audience	7
Associated Document(s)	7
Naming	7
Supported Facilities	8
Programming Overview	9
Programming Language	9
Java / Dot Net / Managed storage	9
Low level definition of 'C' style	10
64 bit and 32 bit systems	10
Background	10
DLL Location	10
Native Programs (C++, Delphi, etc.)	11
C# Programs (Interpreted)	11
Document Layout	12
Getting Started	14
Installation	14
USB Device (Genoa)	14
USB Driver Program	15
DLL Locations under Windows	15
PCI Card	15
Operation	16
OpenMHE	17
OpenSpecificMHE	18
EnableInterface	19
DisableInterface	19
CurrentValue	20
PayOut	20
LastPayStatus	21
CurrentPaid	22
SetDispenseQuantity	23
PaySpecific	24
IndicatorOn / IndicatorOff	24
SwitchOpens / SwitchCloses	25
Getting Started Code Examples	26
Currency Accept	26
Currency Payout	27
Indicator Example	28
Switch Example	28
Full Application	29
Background	29
Controlling an Acceptor	30
Acceptor Enable / Disable	30
AcceptorBlock structure / object	30

Controlling a Dispenser	32
<i>The Status field</i>	32
<i>DispenserBlock structure / object</i>	32
Other Constants	33
<i>Device Identity Constants</i>	33
CurrentUpdates (1.10.4)	35
ReadAcceptorDetails	36
WriteAcceptorDetails	36
ReadDispenserDetails	37
WriteDispenserDetails	37
<i>Dispenser Value Reassignment</i>	38
<i>Dispenser Dump</i>	38
Single Note Head Escrow	39
Escrow Overview	39
Escrow system usage	39
EscrowEnable	40
EscrowDisable	40
EscrowThroughput	41
EscrowAccept	41
EscrowReturn	42
Multi note Extended Escrow (1.12.6)	43
Overview	43
<i>Accepting Notes</i>	43
<i>Returning Notes</i>	43
<i>Keeping Notes</i>	43
EscrowControlBlock	44
Operation	46
Abnormal Situations	47
ReadEscrowBlock	48
EscrowCommand	49
Bar Codes	50
BarcodeEnable	51
BarcodeDisable	51
BarcodeInEscrow / BarcodeInEscrowExt	52
BarcodeStacked / BarcodeStackedExt	53
BarcodeAccept	54
BarcodeReturn	54
Barcode Printing	55
BarcodePrint	55
BarcodePrintStatus	56
Cashless Processing	57
Background	57
<i>Abnormal Processing</i>	58
CashlessReadData	58
CashlessEnable	59
CashlessDisable	59
SubmitTicket	60
CashlessRequestCredit	60
CashlessRefuseCredit	61

CashlessTakeCredit.....	61
CashlessCancelCredit	62
CashlessPayCredit	62
CashlessReset.....	63
Meters / Counters.....	64
<i>Mechanical Meters (1.12.4)</i>	64
CounterIncrement	64
CounterCaption.....	65
CounterRead.....	66
ReadCounterCaption	66
CounterDisplay	67
MeterStatus.....	67
MeterSerialNo	68
E²Prom.....	69
E2PromReset.....	69
E2PromWrite.....	69
E2PromRead	70
Paylink Event Queue	71
Introduction	71
NextEvent	72
NextAcceptorEvent NextDispenserEvent NextSystemEvent	73
Utility Functions.....	74
CheckOperation (1.11.x).....	74
SetDeviceKey	75
SerialNumber	75
FirmwareVersion.....	76
USBDriverStatus.....	77
USBDriverExit	77
IMHEIConsistencyError.....	78
Engineering Support	79
WriteInterfaceBlock.....	79
ReadInterfaceBlock.....	80
DES security (25-12-1).....	81
Background.....	81
Key Exchange.....	81
Des Lock.....	82
DESSetKey.....	82
DESLockSet.....	83
DESLockClear	83
DESStatus	84
Appendix 1 - RawEvent Data for errors	85
Simple RawEvent data error code handling.....	85
MDB RawEvent Data Error Codes.....	85
BNR RawEvent Error Codes.....	85
<i>Acceptor</i>	85
<i>Dispenser</i>	86
BCR / CR RawEvent Error Codes.....	86
Disclaimer.....	89

Revision History

Version	Date	Author	Description
0.0 - Draft	5 th Feb 03	D. Bush A. Graham	Initial description document.
0.1 - Draft	16 th Feb 03	D. Bush	Detail corrections (Bug Fixes)
0.2 - Draft	28 th Feb 03	D. Bush	Changes to Coin Path handling
0.3 - Draft	10 th Apr 03	D. Bush	Minor change to SystemStatus
0.4 - Draft	30 th Apr 03	D Bush	Further Changes to Coin Path Handling
1.0	14 th Oct 03	D Bush	Addition of Meters Various clarifications
1.1	24 th Nov 03	D Bush	New Meter Functions Changes to details on dispensers
1.2	3 rd Dec 03	D Bush	New E2Prom Functions
1.3	2 nd Apr 04	D. Bush	Various Bug Fixes - new constants
1.4	9 th Aug 05	D Bush.	Sections added on: <ul style="list-style-type: none"> • Escrow functions. • Event Queue • Barcode functions
1.5	8 th Mar 06	D Bush	Document structure revised Added a number of Usage Details sections 1.10.x Functions detailed
1.6	13 th Nov 06	A Tainsh	Rewritten the Coin Routing description
1.7	11 th Sep 07	D Bush	Added description of multiple unit support.
1.8	11 th Mar 09	D Bush	Added 1.11.x Functions.
1.9	7 th Oct 10	D Bush	Added Future DES Facilities
1.10	2 nd Mar 11	D Bush	Updated to reflect other environments
1.11	24 th Dec 11	D Bush	Added 1.12.3 Recycler details
1.12	24 th Apr 12	D Bush	Added more recyclers Added NextxxxEvent calls Added mechanical Meters Added Barcode details to Acceptor block
1.13	3 rd Feb 13	D Bush	Details moved to Milan System Manual
1.14	24 June 13	D Bush	Added Precise payments and Extended Escrow
1.15	2 August 13	D Bush	Minor updates
1.16	17 August 16	D Bush	Added Multi Note Escrow and Cashless
1.17	28 April 20	D Bush	Minor updates
1.18	28 March 22	D Bush	Cashless and General updates

Introduction

Purpose of Document

This document describes the software interface to the AES Intelligent Money Handling Equipment Interface (Milan / Paylink), as seen by a software engineer writing programs for a PC.

Intended Audience

The intended audience of this document is the software engineers who will be writing software on the PC that will communicate with the Paylink unit itself or will read the monetary information or diagnostic information provided by the card.

Associated Document(s)

This document is one of a pair that together cover creating and using a Paylink system. This document is written for the use of programmers and covers the details of how to write the programs that interface to Paylink.

The companion document “Milan / Paylink System Manual” is written for the use of the person who is possibly not a programmer, but is concerned with designing and setting up the system centred on a Paylink unit. That document also covers the configuration settings that are used to describe the units connected to Paylink, and the way in which such units are controlled.

Note that *this* document does not stand alone as a description of Paylink. In order to understand how the functionality described in this document operates it is essential the programmer is familiar with the contents of the “Milan / Paylink System Manual” document.

Naming

The system described here has a few names. This section attempts to explain them.

AES	Aardvark Embedded Solutions - us.
IMHEI	Intelligent Money Handling Interface Equipment. This was the original name for the project, This was however difficult to say, and so was replaced in common use by Milan. It remains in the names in of the header files etc.
Milan	This was originally the name of the first hardware build. It has however become the name of the overall project. Most documents from AES talk about Milan to cover the whole family of products that are used with this API
Paylink	This is the name of the USB module(s) made and sold by Money Control under licence from AES. There are at present three versions of Paylink: Paylink The original, metal cased version. Paylink Lite 2 A much smaller, plastic cased version with all the drivers operating on the PC.. uPaylink (Micro Paylink) a PC software only version, for use with Money controls USB peripherals.
PCI Card	This is the original obsolete hardware unit. It was known as Milan a long time ago, but this is its current name,

Supported Facilities

It should be noted that this document cover all versions of the Milan / Paylink system, even those versions that are not yet generally available.

Where a facility may not be available with the version that you are running, the topic titles are suffixed with a version indication in brackets

Programming Overview

Programming Language

When Paylink was created, the majority of the users of Paylink used the C++ language, and so the primary DLL interface uses C / C++ interface styles. This document therefore defaults to describing all the functions and data structures using C++ terminology.

The DLL entry point definitions are provided in a Visual Studio compatible library, **AesImhei.lib**. A translation of this suitable for Borland Code Builder is available as **BorlandAesImhei.lib**.

However, Paylink is entirely independent of the language / environment used and so the Paylink package also includes additional interfaces for new development environments - where relevant, terminology for those is also included.

32 bit Visual Basic (VBA and VB6) and Delphi are capable of accessing these C++ structures and functions directly and so header files for these are provided for inclusion into the project.

Java, C# and VB.NET cannot easily access these and so interface libraries are included in the distribution that implement very similar classes in a manner compatible with the environments.

Java / Dot Net / Managed storage

Java, C# and VB.Net are all object oriented languages that use managed storage.

Paylink provides interface libraries (AesImhei.java / AesImhei.net.dll) for these languages. The interface provided has the following features in all these languages:

- The interface provides a single object, AesImhei, which has no properties and a set of static methods. These methods map directly onto the functions described in this document. As static methods, they are called by prefixing the function name with AesImhei, rather than creating an object of this type.
- This AesImhei object defines in turn a set of objects, which have properties, but no methods. These objects map directly onto the data structures described in this document.
- Where this document refers to a 'C' style array (see below), the AesImhei object creates a managed array of the object so that access to these elements can be carried out as normal.
- Where the base functions in the C++ interface use "char*" strings (see below), these strings are automatically converted by the interface library to and from native string types.
- All objects in these languages are always "Call by Reference", in 'C' this has to be indicated by the inclusion of a '*' in the function definitions in this manual. This '*' should be ignored by programmers in these languages.

Low level definition of 'C' style.

For people working in other languages wishing to understand the function descriptions and data structure excerpts in this document and the original descriptions in the definitive C / C++ header file, the following type are used in both:

<code>int</code>	A 32 bit signed quantity (in other languages <i>Int32</i> or <i>Integer</i> .) Previous Paylink releases used <i>long</i> but this now is tending to denote a 64-bit quantity.
<code>char</code>	An 8 bit quantity. This will be used to hold numbers in the range 0 to 127. This is sometimes Byte or SByte.
<code>char*</code>	A 'C' style string. In detail, this is the memory address of an area containing successive <i>chars</i> (bytes) with the end denoted by a char with a value of zero.

Arrays declarations in 'C' are of the form:

`Type Variable[Length]`

where Type is one of the above, and Length is the number of items in the array.

Where an array is used as a part of a structure, C++, and hence the underlying interface DLL, has the storage described actually within the structure at the point of the declaration.

When functions in this document use structures, they always suffix the structure name with a *. This accomplishes "Call by Reference" where the value passed is the memory address of the 1st byte of the structure.

64 bit and 32 bit systems.

Background

The processor chip in a PC can run in two different modes, one with 32 bit addressing and one with 64 bit addressing. These two modes have completely different instructions sets; code that runs in one mode is completely meaningless to the chip if it is set in the other mode.

A 64 bit OS runs its internal operations with the chip in 64-bit mode. When it is starting a program, it checks the header of the .exe file, and if the header is marked 32-bit, it switches the chip to 32-bit mode before starting to execute the program.

Once execution has started, the mode of the chip cannot be changed; all operations in the program will be in the same mode. The important consequence is that all DLLs must be in the same mode.

This leads to an interesting problem with system DLLs - a 64 bit operating system has two complete sets of system DLLs, one compiled in 64 bit mode, the other in 32 bit mode, and each of a pair of DLLs has the same name.

DLL Location

The standard location for a system DLL was always Windows\System32 so a lot of existing code uses that as the known location of a system DLL, so all the 64-bit mode DLLs are held there, to quote Microsoft: "Windows x64 has a directory System32 that contains 64-bit DLLs. Thus native processes with a "bitness" of 64 find "their" DLLs where they expect them: in the System32 folder. A second directory, SysWOW64, contains the 32-bit DLLs. The file system redirector does the magic of hiding the real System32 directory for 32-bit processes and showing SysWOW64 under the name of System32."

So what this is all saying is that, as the Paylink utility software is all 32 bit, a DLL actually located in C:\Windows\SysWOW64 will be reported, by Windows itself, as in C:\Windows\System32 whenever a Paylink program looks for it - and they will (erroneously) report the DLL as being located in C:\Windows\System32.

As we distinguish the bitness of our DLLs in the name, we *recommend* that both DLLs are put into Windows\System where they will be found by all programs.

Native Programs (C++, Delphi, etc.)

With Paylink we provide two DLLs, but we distinguish them by name. AESImhei.dll is compiled into 32-bit code, AESImhei64.dll is compiled into 64-bit code. People writing a C++ (or any other native mode) program can choose the DLL name that matches the mode of their application.

C# Programs (Interpreted)

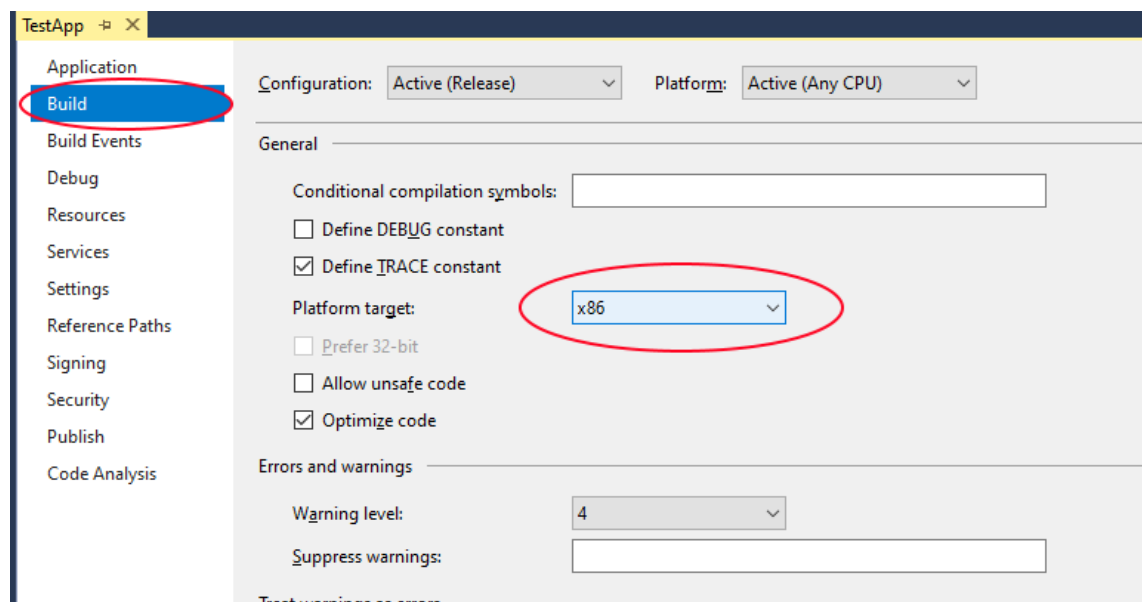
A separate document concentrates on describing the Dot Net specifics in a Paylink release.

In brief, C# programs / DLLs are interpreted and a 64 bit version of Windows has two separate interpreters available; a 32 bit one and a 64 bit one. The input to these two interpreters is not dependant on the mode of the processor – it is one single format, and so most C# projects are marked as Neutral. With 64 bit Windows, these Neutral programs cause the 64-bit interpreter to be run as it is the default. You then have to ensure that when you use the native DLL, you choose the correct one, either AESImhei.dll or AESImhei64.dll, to match the interpreter that has been chosen to run your application.

It is possible to mark a C# program as only to be run by the 32-bit (Win32 / x86) interpreter, and if you include AESImhei.Net.dll in your project then this is a required step if you're running 64-bit Windows.

If you are running 64-bit Windows on both the development and target systems, then it would make sense to use AESImhei64.Net.dll as that will automatically be the default configuration.

The Paylink SDK as released includes projects for TestApp and TestApp64 - which have the platform target already selected, e.g.



Document Layout

This document itself is split into a number of sections. Within each section, there are three parts.

- **Operational Overview.**
Where the way in which this area is intended to work is explained.
- **Function Definitions.**
Where you will find exact details on each function call.
- **Usage Details.**
This gives details on exactly how the Milan / Paylink system operates.

The first three sections are intended to reflect different levels of complexity at which an initial application programmer may wish to use the interface.

1. Getting Started

These are the minimum set of “vanilla” functions that may be used to get a working *demonstration* program running.

Using these calls alone; the software engineer can write a working program and get a feel for the ease with which he can now communicate with the Money Handling Equipment attached to his system.

2. Full Application

These build on the set of functions provided within the “Getting Started” section. They add functionality that can determine the *status* of the peripherals attached to the interface card.

By these status analysis functions, the application programmer could determine (say) the exact reason that an attempted payout failed and then notify either an engineer or a cash collector.

3. Utility Functions

These miscellaneous functions are concerned with the administration of the application system.

4. Note Reader Escrow

Here you will find functions that enable the escrow feature provided by note acceptors to be easily used.

5. Meters / Counters

This section is concerned with the support of the SEC meter, a small external unit that allows audit numbers to be maintained

6. E2Prom

The Paylink units incorporate E²Prom storage for internal configuration storage. Some of this is made available to the PC programmer.

7. Barcode Reading

Here you will find functions that enable the barcoded ticket features provided by some note acceptors to be easily used.

8. Barcode Printing

These functions are used by the Paylink units to support a "Ticket Printer", which will produce barcoded tickets.

9. DES security (25-12-1 onwards)

These functions are those that enable an application to apply and check a security lock to a DES Paylink.

Using these calls, the software engineer can ensure that a DES Paylink and set of DES peripherals cannot be tampered with, or used by any other program / PC.

10. Engineering Support

These functions provide full-blown diagnostics and reconfiguration facilities.

Getting Started

Installation

USB Device (Genoa)

All three Milan / Paylink devices use the same USB interface chip, in different mode. This is a standard USB 1.1 peripheral.

Installation of the **Windows** driver can happen as with any USB peripheral: when the unit is detected Windows will automatically access the Internet and download.

Currently the Paylink system is released as a Windows installation file. This automatically carries out the following operations, which you may wish to undertake manually on your target system.

- The interfaces **AESImhei.dll** and **AESImhei64.dll** need to be copied to the hard disc. Ideally this will be to a single shared location below the C:\Windows folder. We recommend the use of the C:\Windows\System folder, but alternatives are detailed below in the section on DLL locations.
- The High Level driver program **Paylink.exe** needs to be copied from the installation CD to a convenient folder and an entry made in the Startup folder / System Registry to run this at system boot. As detailed in the "Milan / Paylink System Manual", the system design of Paylink and its support software expects this program to be running continually.
- For Java and dot Net, the appropriate interface specific DLL has to be copied to an appropriate location - again see below for DLL locations.

For **Linux** systems, the release is provided as an archive, containing the source of all the required PC software. This consists of three main elements:

- The low level USB interface library "libusb". This is the standard library, unchanged by Aardvark, but provided so as to ease the installation process.
- FTDI specific drivers "libftdi" that allow the Aardvark code to easily access the libusb library. Again Aardvark has modified none of the code.
- The shared library interfaces, the USB driver programs and the examples.

Complete details on how to manage this are provided in a separate "Linux User Guide"

Please note that all the installation scripts are design to be run with full root privilege. (i.e. with **sudo** or after an **su** command.)

USB Driver Program.

As detailed in the “Milan / Paylink System Manual”, the supplied USB driver program Paylink or AESCDriver has to be run, and should be regarded as a system service and unconditionally started at system boot.

Although it is *possible* to stop and start this program, that causes exception processing to be undertaken and may cause the interface presented to the PC to change drastically and is not recommended in any sort of production environment.

Note that the system should not be regarded as usable for 20 seconds after a Paylink reset, or for 10 seconds after a USB driver program (re-)start.

DLL Locations under Windows

The AESImhei.dll (or AESImhei64.dll) DLL needs to be placed into a folder that is in the search path for all application that will use it. The search order in modern Windows systems (that have not been tailored) follows, so the DLL needs to go in one of these places:

1. The folder from which the application will be loaded.
2. The system folder. - On a 32-bit system this is C:\Windows\System32\ for AESImhei.dll, on a 64-bit system this is C:\Windows\SysWOW64\ for the 32-bit AESImhei.dll and C:\Windows\System32\ for AESImhei64.dll.
3. The “C:\Windows\System\” folder - as this is common to all “bitnesses” & systems, we recommend this location.
4. The “C:\Windows\” Folder.
5. The “Start In” folder from the short cut of the application.
6. The folders that are listed in the PATH environment variable.

PCI Card

The obsolete PCI card was a standard PCI interface card which has the normal Windows Plug 'n' Play automatic installation facilities.

Note that PCI Cards do not support version 1.11.1 and later, do not have any interface to a Linux system and are not supported in 64-bit versions of Windows.

Operation

Whichever type of Milan / Paylink product used, the driver software (possibly in conjunction with the Paylink unit) is responsible for all communication with the peripherals,

It handles the *event* based protocols, and uses the results to update a set of *state* tables.

The underlying concept behind the use of state tables is that all activity causes counters to be incremented. The application programmer reads out the totals at the time the application starts, and then compares these with the current totals. Peripheral activity will cause these totals to increment, subtracting the old, saved value from the current value enables the application to determine the value inserted by the customer.

Using state tables on the PC in this way allows the programmer to be unconcerned with hardware response times. Although the *state* tables have to be periodically examined to see if anything has changed, there is never any requirement that this is done quickly, and the programmer does not have to be concerned that the OS may suspend his program for significant periods. Regardless of how long the program spends between examinations, the system will function perfectly and no money insertion or payout will be missed.

The following section describes the minimum set of function calls needed to implement a useful system. Using the functions described within this section, one can provide a fully working system, with credit and payout capability, as well as a number of indicators and switches.

OpenMHE

Synopsis

This call is made by the application to open the “Money Handling Equipment” Interface.

```
int OpenMHE (void) ;
```

Parameters

None

Return Value

If the Open call succeeds then the value zero is returned.

In the event of a failure one of the following standard windows error codes will be returned, either as a direct echo of a Windows API call failure, or to indicate internally detected failures that closely correspond to the quoted meanings.

Windows Error Number	Linux Error Number	Suggested string for English decoding	Microsoft Mnemonic	Retry
13	1	The DLL, application or device are at incompatible revision levels	ERROR_INVALID_DATA	No
20	4	General system error	ERROR_BAD_UNIT	No
21	3	Paylink has not yet started	ERROR_NOT_READY	Yes
31	2	Driver program not running	ERROR_GEN_FAILURE	Yes
170	170	The USB link is in use	ERROR_BUSY	Yes
1167	1167	No Paylink unit is connected	ERROR_DEVICE_NOT_CONNECTED	Yes

Remarks

1. As with any USB system, there is a noticeable time for the USB communications to start. This may cause error returns labelled “Yes” under Retry in the above table. This indicates that the call to **OpenMHE** should be retried periodically until it has failed for at least 20 consecutive seconds before deciding that the interface is actually inoperative.
2. Whereas an Open service normally requires a description of the item to be opened (and returns a reference to that Item) normally there is only one IMHE Interface unit in a system. Hence any “Open” call refers to that single item.
3. Even following this call, all the money handling equipment will be *disabled* and hence rejecting all currency inserted and ignoring switches until the successful execution of a call to **EnableInterface**.

OpenSpecificMHE

Synopsis

This call is made by the application to open or to switch to one of the multiple “Money Handling Equipment” Interfaces installed on the PC.

Details on how a system works with multiple Paylinks are given in a later section.

C++

```
int OpenSpecificMHE (char SerialNo[8]);
```

C#, VB, Java

```
int OpenSpecificMHE (string SerialNo);
```

Parameters

None

Return Value

If the Open call succeeds then the value zero is returned.

In the event of a failure the same standard windows error codes are returned as for **OpenMHE**.

Remarks

1. Every Paylink requires a unique instance of the USB driver program to be running. If there is no driver for the Paylink whose Serial Number is quoted, then the function returns 31 (ERROR_GEN_FAILURE).
2. As the default serial number for Paylink unit is “AE000001”, the **OpenMHE** call is equivalent to the call **OpenSpecificMHE("AE000001")**, used with a driver program specified to communicate with Serial Number AE000001.
3. This call may be issued repeatedly with no ill effects. Each call will serve to swap all the other calls in this document to the specified unit.

EnableInterface

Synopsis

The **EnableInterface** call is used to “turn on” the Milan system. This would be called when the system is initialised and ready.

Until this call is made, none of the functions concerning the operation of the Paylink unit will work. This includes such thing **CheckOperation** and items such as switches and meters function.

If you wish to start running with acceptance disabled, this call should still be made and money acceptance disabled as in the section *Acceptor Enable / Disable* on page 30 below in the part headed *Controlling an Acceptor*.

void EnableInterface (void) ;

Parameters

None

Return Value

None

Remarks

1. Normally the application will initialise the saved values of all the information it is monitoring before this call.
2. This must be called following the call to **OpenMHE** before any coins / notes will be registered.

DisableInterface

Synopsis

The **DisableInterface** call is provided for completeness, and *completely* “closes down” the Milan system, prior to a system close down. *Note that **all** Paylink interface items including switches, peripheral reporting and meters also cease functioning at this point.*

To just disable money acceptance see the section *Acceptor Enable / Disable* on page 30 below in the part headed *Controlling an Acceptor*.

void DisableInterface (void) ;

Parameters

None

Return Value

None

Remarks

1. There is no guarantee that a coin or note cannot be successfully read after this call has been made, a successful read may be in progress. The read will not however be reported!

CurrentValue

Synopsis

Determine the current monetary value that has been accepted

The **CurrentValue** call is used to determine the total value of all coins and notes read by the money handling equipment connected to the interface.

int CurrentValue (void) ;

Parameters

None

Return Value

The current value, in the lowest denomination of the currency (i.e. cents / pence etc.) of all coins and notes read.

Remarks

1. The value returned by this call is never reset, but increments for the life of the interface card. Since this is a 32 bit integer, the card can accept £21,474,836.47 of credit before it runs into any rollover problems. In most jurisdictions this value is expected to exceed the life of the unit.
2. It is the responsibility of the application to keep track of value that has been used up and to monitor for new coin / note insertions by increases in the returned value.
3. Note that this value should be read following the call to **OpenMHE** and before the call to **EnableInterface** to establish a starting point before any coins or notes are read.

PayOut

Synopsis

The **PayOut** call is used by the application to instruct the interface to pay out coins (or notes).

void PayOut (int Value) ;

Parameters

Value

This is the value, in the lowest denomination of the currency (i.e. cents / pence etc.) of the coins and notes to be paid out.

Return Value

None

Remarks

1. This function operates in value, not coins. It is the responsibility of the Paylink interface to decode this and to choose how many coins (or notes) to pay out, and from which device to pay them.

LastPayStatus

Synopsis

The PayStatus call provides the current status of the payout process.

```
int LastPayStatus (void) ;
```

Parameters

None

Return Values.

Value	Meaning	Mnemonic
0	The interface is in the process of paying out	PAY_ONGOING
1	The payout process is up to date	PAY_FINISHED
-1	The dispenser is empty	PAY_EMPTY
-2	The dispenser is jammed	PAY_JAMMED
-3	Dispenser non functional	PAY_US
-4	Dispenser shut down due to fraud attempt	PAY_FRAUD
-5	The dispenser is blocked	PAY_FAILED_BLOCKED
-6	No Dispenser matches amount to be paid	PAY_NO_HOPPER
-7	The dispenser is inhibited	PAY_INHIBITED
-8	The internal self-checks failed	PAY_SECURITY_FAIL
-9	The hopper reset during a payout	PAY_HOPPER_RESET
-10	The hopper cannot payout the exact amount	PAY_NOT_EXACT
-11	This hopper does not really exist.	PAY_GHOST
-12	Waiting on a valid key exchange	PAY_NO_KEY

Remarks

1. Following a call to **PayOut**, the programmer should poll this to check the progress of the operation.
2. If one or more of multiple hoppers has a problem, the Paylink will do the best it can. If it cannot pay out the entire amount, the error status returned will reflect the problem with the *last* attempt.

CurrentPaid

Synopsis

The CurrentPaid call is available to keep track of the total money paid out because of calls to the PayOut function.

int CurrentPaid (void) ;

Parameters

None

Return Value

The current value, in the lowest denomination of the currency (i.e. cents / pence etc.) of all coins and notes ever paid out.

Remarks

1. The value that is returned by this function is updated in real time, as the money handling equipment succeeds in dispensing coins.
2. The value that is returned by this call is never reset, but increments for the life of the interface card. It is the responsibility of the application to keep track of starting values and to monitor for new coin / note successful payments by increases in the returned value.
3. While a payout is in progress, the value *can* reduce, if for instance a note that has not been taken is retracted.
4. Note that this value can be read following the call to **OpenMHE** and before the call to **EnableInterface** to establish a starting point before any coins or notes are paid out.

SetDispenseQuantity

Synopsis

The **SetDispenseQuantity** call will set the given quantity as the number of coins (notes) to be dispensed from a specific dispenser on the next call to **PaySpecific** (). The next section “**Full Application**” explains about dispensers.

```
int SetDispenseQuantity( int Index,  
                        int Quantity,  
                        int Value) ;
```

Parameters

1. Index
This parameter specifies the dispenser that is having the count set.
2. Quantity
This sets the quantity of coins (notes) to be dispensed from the indicated dispenser.
3. Value
This is provided as a cross check, and **must** be the value of the coin / notes dispensed by this dispenser.

Return Value

If the dispenser referenced is valid, and is set up for coins (notes) of the specified value, the return value is value of the resultant payout (i.e. Quantity * Value). If there is a problem in the specification, then zero is returned.

Remarks

1. Once a quantity has been set by use of this call, it remains set through all other Paylink interface calls until cleared as a side effect of a **PaySpecific** () call.
2. Although both are not necessary, both the Index and the Value parameters are required as a security check.
3. A non-zero return indicates only that the payout will be attempted - no reference is made to the operability of the dispenser.

PaySpecific

Synopsis

The **PaySpecific** call takes no parameters. It causes Paylink to attempt to pay out all the coins (notes) specified by earlier calls to **SetDispenseQuantity()**.

This operation proceeds exactly as with a payout started by the Payout command so far as monitoring the on-going payout and determining the result. The only difference is in the way that Paylink determines how many of each denomination to try to pay out.

```
int PaySpecific () ;
```

Parameters

None

Return Value

The total value of the payout being attempted.

Remarks

1. The only differences between the progress of a payout started by **PaySpecific()** and one started by the traditional **PayOut ()** call is the quantity of the different coins (notes) chosen, and the fact that there is no "fall over" to a lower value dispenser if a higher value dispenser is, or becomes, empty.
2. As with **PayOut()**, progress is monitored by repeated calls to **LastPayStatus()** waiting for PAY_ONGOING to change. Again, as with a pay-out started by **PayOut()**, the total value paid can be monitored by calls to **CurrentPaid()** and the coins (notes) paid for each dispenser found / monitored using the Count field of the Dispenser blocks
3. Having transferred the counts set by **PaySpecific()** to the Paylink unit for this pay out, the counts are then cleared.

IndicatorOn / IndicatorOff

Synopsis

The IndicatorOn / IndicatorOff calls are used by the application to control LED's and indicator lamps connected to the interface.

```
void IndicatorOn (int IndicatorNumber) ;  
void IndicatorOff (int IndicatorNumber) ;
```

Parameters

IndicatorNumber

This is the number of the Lamp that is being controlled.

Return Value

None

Remarks

1. Although the interface is described in terms of lamps, any equipment at all may in fact be controlled by these calls, depending only on what is physically connected to the interface card.

SwitchOpens / SwitchCloses

Synopsis

The calls to **SwitchOpens** and **SwitchCloses** are made by the application to read the state of switches connected to the interface card.

```
int SwitchOpens (int SwitchNumber) ;  
int SwitchCloses (int SwitchNumber) ;
```

Parameters

SwitchNumber

This is the number of the switch that is being controlled. In principle the API can support 64 switches, though note that not all of these may be supported by any particular hardware unit.

Return Value

The number of times that the specified switch has been observed to open or to close, respectively.

Remarks

1. The convention is that at initialisation time all switches are open, a switch that starts off closed will therefore return a value of 1 to a SwitchCloses call immediately following the OpenMHE call.
2. The expression (SwitchCloses(n) == SwitchOpens(n)) will always return 0 if the switch is currently closed and 1 if the switch is currently open.
3. Repeat pressing / tapping of a switch by a user will be detected by an increment in the value returned by SwitchCloses or SwitchOpens.
4. The user only needs to monitor changes in one of the two functions (in the same way as most windowing interfaces only need to provide functions for button up or button down events)
5. The inputs are debounced. The unit reads all 16 inputs every 2 milliseconds. If we detect a change, we then require the next two reads to give exactly the same pattern before reporting the change. This means that a simple "electronic" input change will be reported between 4 and 6 milliseconds of it occurring.

Getting Started Code Examples

The following 'C' code fragments are intended to provide clear examples of how the calls to the Paylink are designed to be used:

Each function will provide the central processing for a small command line demonstration program.

Currency Accept

```
void AcceptCurrencyExample(int NoOfChanges)
{
    int LastCurrencyValue ;
    int NewCurrencyValue ;

    int OpenStatus = OpenMHE() ;

    if (OpenStatus != 0)
    {
        printf("IMHEI open failed - %ld\n", OpenStatus) ;
    }
    else
    {
        // Then the open call was successful
        // Currency acceptance is currently disabled
        LastCurrencyValue = CurrentValue() ;

        printf("Initial currency accepted = %ld pence\n",
               LastCurrencyValue) ;

        EnableInterface() ;

        printf("Processing %d change events\n", NoOfChanges) ;
        while (NoOfChanges > 0)
        {
            Sleep(100) ;

            NewCurrencyValue = CurrentValue() ;
            if (NewCurrencyValue != LastCurrencyValue)
            {
                // More money has arrived (we do not care where from)
                printf("The user has just inserted %ld pence\n",
                       NewCurrencyValue - LastCurrencyValue) ;
                LastCurrencyValue = NewCurrencyValue ;
                --NoOfChanges ;
            }
        }
    }
}
```

Currency Payout

```
void PayCoins(int NoOfCoins)
{
    int OpenStatus = OpenMHE() ;

    if (OpenStatus != 0)
    {
        printf("IMHEI open failed - %ld\n", OpenStatus) ;
    }
    else
    {
        // Then the open call was successful
        // The interface is currently disabled
        EnableInterface() ;

        PayOut(NoOfCoins * 100) ;
        while (LastPayStatus() == 0)
        {
        }
        if (LastPayStatus() < 0)
        {
            printf("Error %d when paying %d coins\n",
                   LastPayStatus(), NoOfCoins) ;
        }
        else
        {
            printf("%d coins paid out\n", NoOfCoins) ;
        }
    }
}
```

Indicator Example

```
void LEDs(void)
{
    int OpenStatus = OpenMHE() ;
    char Loop ;

    if (!OpenStatus)
    {
        EnableInterface() ;

        for (Loop = 0 ; Loop < 8 ; ++Loop)
        {
            IndicatorOn(Loop) ;
            Sleep(1000) ;
        }

        for (Loop = 0 ; Loop < 8 ; ++Loop)
        {
            IndicatorOff(Loop) ;
            Sleep(1000) ;
        }

        DisableInterface() ;
    }
}
```

Switch Example

```
void LEDs(void)
{
    int OpenStatus = OpenMHE() ;
    char Loop ;

    if (!OpenStatus)
    {
        EnableInterface() ;

        for (Loop = 0 ; Loop < 8 ; ++Loop)
        {
            printf("Switch %d is currently %s\n", Loop,
                SwitchCloses(Loop) == SwitchOpens(Loop) ?
                "Open" : "Closed") ;

            printf("It has closed %d times!\n", SwitchCloses(Loop)) ;
        }

        DisableInterface() ;
    }
}
```

Full Application

Background

When implementing a full application implementation, tighter control over the behaviour and response of the individual acceptors and hoppers is frequently necessary, for such purposes as routing coins to hoppers and cashboxes and emptying hoppers. Some more details on these operations are given at the end of this section.

The data retrieval functionality is achieved by reading the control blocks for the acceptors (with **ReadAcceptorDetails**) and possibly hoppers (with **ReadDispenserDetails**) at initialisation time and then continually checking the current contents of these against saved copies. To aid in this process the **CurrentUpdates** function guarantees that; if it returns an unchanged value then *nothing* in *any* control block will have changed.

Most of the control functionality is achieved by reading a data structure from the API, modifying it as appropriate or necessary and writing it back. Four functions are involved: **ReadAcceptorDetails**, **ReadDispenserDetails**, **WriteAcceptorDetails** & **WriteDispenserDetails**.

All these functions identify the individual units by a sequence number, in the range 0...N-1. The programmer should not assume that any particular unit is present at any particular number; the numbers are assigned dynamically and are liable to change from run to run.

To find the particular unit of interest, the programmer should scan number from 0 up, looking for a match on the structure members.

Controlling an Acceptor

For an acceptor, the relevant control block usually involves matching on the **Unit** field. Although this is defined as single 32 bit number, it is created by concatenating four 8 bit values. The program will usually only be interested in distinguishing the coin and note acceptors, which are distinguished by values in the top 8 bits. For this purpose two 'C' macros are defined, **IS_COIN_ACCEPTOR(Unit)** and **IS_NOTE_ACCEPTOR(Unit)**, see below, which can easily be translated into other languages.

The **AcceptorBlock** contains various fields that are relevant to the acceptor as a whole, and an embedded array of **AcceptorCoins**, which contain fields relevant to individual coins (or notes.)

To control an acceptor, the correct method is to find the acceptor is question with **ReadAcceptorDetails()** (see below), modify one or more field and then write the modified data back using **WriteAcceptorDetails()** (see below)

Most fields within the **AcceptorBlock** are self-explanatory, or are adequately described by the comment included in the header file included below.

A few fields are worthy of a more detailed explanation are:

Status: Each bit of this field has a specific use. The bits are itemised below and are named **ACCEPTOR_XXX** where xxx is the usage.. Note that **ACCEPTOR_INHIBIT** is, uniquely, set by the application and read by Paylink.

NoOfCoins This specifies the number of entries in the **Coin** array that are used. Later entries are neither written to nor read.

Fields with names involving "Path" are concerned with bill / coin routing, a detailed description of which is given in the section on routing in the "Milan / Paylink System Manual".

Acceptor Enable / Disable

Enabling / disabling money acceptance is performed as with any other acceptor control. The specific operation is to ensure that the **bit ACCEPTOR_INHIBIT** is set in the **Status** field (usually by using the OR operation) before writing back the modified data.

When reading the **AcceptorBlock** for an acceptor that has been disabled, this bit will be set. To enable an acceptor it will be necessary to clear this bit (usually by using an AND operation) before writing back the modified data.

It is acceptable practice to disable or enable all acceptors in the system, without paying any attention to any other information about them.

AcceptorBlock structure / object

Constants for Acceptors

Name	Meaning
ACCEPTOR_DEAD	No response to communications for this device
ACCEPTOR_DISABLED	The acceptor is currently disabled (for any reason)
ACCEPTOR_INHIBIT	Set by the application to disable the acceptor
ACCEPTOR_FRAUD	Fraud attempt has been detected
ACCEPTOR_BUSY	the Device is reporting itself as busy
ACCEPTOR_FAULT	The Milan unit cannot communicate with the device
ACCEPTOR_NO_KEY	The Milan unit needs to acquire a DES key from the unit
MAX_ACCEPTOR_COINS	The absolute Maximum coins or notes handled by any device

Fields / properties for the AcceptorBlock Object

Field Type	Name	Meaning
int	Unit	Specification of this unit
int	Status	AcceptorStatuses - zero if device OK
int	NoOfCoins	The number of different coins handled
int	InterfaceNumber	The bus / connection
int	UnitAddress	For addressable units
int	DefaultPath	The default routing for coins
int	BarcodesStacked	The total number of barcode tickets stacked by this acceptor
char[4] string	Currency	Main currency code reported by an acceptor
AcceptorCoin	Coin[MAX_ACCEPTOR_COINS]	The coins (only NoOfCoins are set up)
int	SerialNumber	Reported serial number (0 if N/A)
char* string	Description	Device specific string for type / revision / coin set
int	EscrowBarcodeHere	If this is non zero, then the barcode reported by BarcodeInEscrow is from this acceptor

Fields / properties for the AcceptorCoin sub object

Field Type	Name	Meaning
int	Value	Value of this coin
int	Inhibit	If set non zero by the PC: this coin is inhibited
int	Count	Total number read "ever"
int	Path	Set by PC: this coin's chosen output path
int	PathCount	Number "ever" sent down the chosen Path
int	PathSwitchLevel	Set by PC: PathCount level to switch coin to default path
char	DefaultPath	Set by PC: Default path for this specific coin
char	FutureExpansion	Set by PC: for future use
char	HeldInEscrow	count of this note / coin in escrow (usually max 1)
char	FutureExpansion2	for future use
char* string	CoinName	A string, usually as returned from the acceptor, describing this coin

Note that although the routing fields, Path and DefaultPath, are defined as "set by PC", they are, where possible, initialised to the values read from the acceptor.

Controlling a Dispenser

For a dispenser, the relevant control block usually involves matching on the **Value** field as that shows the coin value being used by the Paylink unit, which is the most important distinguishing feature of a dispenser, although for bill recyclers etc. the **UnitAddress** field is often useful.

The **Inhibit** field of a dispenser can be set a value other than zero to prevent Paylink from trying to use that dispenser to satisfy a Payout request. Note that setting this field has no actual effect on the peripheral itself.

The Status field.

There are plans to improve the reporting and control of a Dispenser, but at present all reporting and control takes place through the single **Status** field.

The details around the **Status** field of a dispenser can at first be difficult to grasp:

The field is used to report problems with the dispenser to the application. In general, the dispenser *itself* can only notice a problem when it actually tries to deliver money to the end user.

The result of this is that the dispenser status is probably badly named - it is not the current status, but is instead the result of the last attempt to deliver money. (This is further complicated by the fact that the real time status PAY_US overrides the result while there is a connection problem!)

If an attempt to deliver money fails, the dispenser status is set to one of the error values (where the unit tries to select a meaningful error code).

This status (for the last payout attempt) will stay the same, *no matter what happens in the real world*, until another payout attempt is made, and will only **then** clear if the payout is successful.

(Restarting Paylink can cause this status to be forgotten with the status reverting to PAY_FINISHED - this is in fact misleading. In theory, it might make everything clearer to have an initial status of PAY_UNKNOWN to indicate that no pay has been attempted.)

In addition, certain “exotic” operations on a Dispenser can be triggered by the application writing codes into the **Status** field.

DispenserBlock structure / object

Constants for Dispensers

Name	Meaning
MAX_DISPENSERS	Maximum handled
Coin Count Status Values	
DISPENSER_COIN_NONE	No dispenser coin reporting
DISPENSER_COIN_LOW	Less than the low sensor level
DISPENSER_COIN_MID	Above low sensor but below high
DISPENSER_COIN_HIGH	High sensor level reported
DISPENSER_ACCURATE	Coin Count reported by Dispenser
DISPENSER_ACCURATE_FULL	The Dispenser is full
Special Status Values	
DISPENSER_REASSIGN_VALUE	The Value has just been updated by the application
DISPENSER_VALUE_REASSIGNED	The updated Value has just been accepted by Paylink
DISPENSER_CASHBOX_DUMP	Dump the hopper if you can (for note recyclers only)

DISPENSER_PARTIAL_DUMP	Dump some of the hopper if you can
DISPENSER_DUMP_FINISHED	A Recycler dump has just complete

Fields / properties for the DispenserBlock

Field Type	Name	Meaning
int	Unit	Specification of this unit
int	Status	Individual Dispenser status This takes the same values as PayStatus()
int	InterfaceNumber	The bus / connection
int	UnitAddress	For addressable units
int	Value	The value of the coins in this dispenser
int	Count	Number dispensed according to the hopper records
int	Inhibit	Set to 1 to inhibit Dispenser
int	NotesToDump	Used in conjunction with DISPENSER_PARTIAL_DUMP
int	CoinCount	The number of coins in the dispenser
int	CoinCountStatus	Flags Relating to Coin Count (See above)
int	SerialNumber	Reported serial number (0 if N/A)
char* string	Description	Device specific string for type / revision

Other Constants

The following is the description of the structures / object used by this interface library. For full details of the structures the reader is referred to the various header files / interface libraries distributed in the SDK folder of the Milan software releases.

Device Identity Constants

These constants are ORed together to form the coded device identity that can be extracted from the interface.

Example

As an example, a Money Controls Serial Compact Hopper 2 will have the following device code DP_MCL_SCH2, made up from:

- A device specific code ORed with
- DP_COIN_PAYOUT_DEVICE ORed with
- DP_CCTALK_INTERFACE ORed with
- DP_MANU_MONEY_CONTROLS

This is a device code of **0x01020101**

From this, you can create some simple classification functions, e.g.

```

IS_ACCEPTOR(code)      (code & 0x02000000)
IS_COIN_ACCEPTOR(code) ((code & DP_GENERIC_MASK) == DP_COIN_ACCEPT_DEVICE)
IS_NOTE_ACCEPTOR(code) ((code & DP_GENERIC_MASK) == DP_NOTE_ACCEPT_DEVICE)
IS_PAYOUT(code)        (code & 0x01000000)
```

For the complete list of device specific constants the reader is referred to the AESImhei.h 'C' header file, or its language specific equivalent.

The general components of these identities are:

Name	Value	
DP GENERIC MASK	0xff000000	The mask to extract the following parts
DP COIN ACCEPT DEVICE	0x02000000	
DP NOTE ACCEPT DEVICE	0x12000000	
DP CARD ACCEPT DEVICE	0x22000000	
DP COIN PAYOUT DEVICE	0x01000000	
DP NOTE PAYOUT DEVICE	0x11000000	
DP CARD PAYOUT DEVICE	0x21000000	
These describe the interface via which this device is connected:		
DP INTERFACE MASK	0x00ff0000	The mask to extract the following parts
DP INTERFACE UNIT	0x00000000	
DP CCTALK INTERFACE	0x00020000	
DP SSP INTERFACE	0x00030000	
DP HII INTERFACE	0x00040000	
DP ARDAC INTERFACE	0x00050000	
DP JCM INTERFACE	0x00060000	
DP GPT INTERFACE	0x00070000	
DP MDB INTERFACE	0x00080000	
DP MDB LEVEL 3 INTERFACE	0x00080000	
DP MDB LEVEL 2 INTERFACE	0x00090000	
DP F56 INTERFACE	0x000A0000	
DP CCNET INTERFACE	0x000B0000	
These describe the manufacturer of the device.		
DP MANUFACTURER MASK	0x0000ff00	The mask to extract the following parts
DP MANU UNKNOWN	0x00000000	
DP MANU MONEY CONTROLS	0x00000100	
DP MANU INNOVATIVE TECH	0x00000200	
DP MANU MARS ELECTRONICS	0x00000300	
DP MANU AZKOYEN	0x00000400	
DP MANU NRI	0x00000500	
DP MANU ICT	0x00000600	
DP MANU JCM	0x00000700	
DP MANU GPT	0x00000800	
DP MANU COINCO	0x00000900	
DP MANU ASAHI SEIKO	0x00000A00	
DP MANU ASTROSYSTEMS	0x00000B00	
DP MANU MERKUR	0x00000C00	
DP MANU FUJITSU	0x00000D00	
DP MANU CASHCODE	0x00000E00	
Some Generic Identities		
DP ID003 NOTE	0 DP JCM INTERFACE	DP NOTE ACCEPT DEVICE
DP MDB LEVEL 2	0 DP MDB LEVEL 2 INTERFACE	DP COIN ACCEPT DEVICE
DP MDB LEVEL 3	0 DP MDB LEVEL 3 INTERFACE	DP COIN ACCEPT DEVICE
DP MDB LEVEL 2 TUBE	0 DP MDB LEVEL 2 INTERFACE	DP COIN PAYOUT DEVICE
DP MDB TYPE 3 PAYOUT	0 DP MDB LEVEL 3 INTERFACE	DP COIN PAYOUT DEVICE
DP MDB BILL	0 DP MDB INTERFACE	DP NOTE ACCEPT DEVICE

CurrentUpdates (1.10.4)

Synopsis

Detect updates to the data presented to the API by the firmware.

The fact that the value returned by **CurrentUpdates** has changed prompts the application to re-examine all the variable data in which it is interested.

int CurrentUpdates (void) ;

Parameters

None

Return Value

Technically **CurrentUpdates** returns the number of times that the API data has been updated since the PC system initialised. In practice, only *changes* in this value are significant.

Remarks

1. It is possible that the value could change without any visible data changing.

ReadAcceptorDetails

Synopsis

The ReadAcceptorDetails call provides a snapshot of all the information possessed by the interface on a single unit of money handling equipment.

```
int ReadAcceptorDetails ( int          Number,  
                          AcceptorBlock* Snapshot) ;
```

Parameters

1. Number
The sequence number of the coin or note acceptor about which information is required.
2. Snapshot
A pointer to a program buffer into which all of the information about the specified acceptor will be copied.

Return Value

Non zero if the specified input device exists, Zero if the end of the list is reached.

Remarks

The sequence numbers of the acceptors are contiguous and run from zero upwards.

WriteAcceptorDetails

Synopsis

The **WriteAcceptorDetails** call updates all the changeable information to the interface for a single unit of money accepting equipment.

```
void WriteAcceptorDetails (int          Number,  
                           AcceptorBlock* Snapshot) ;
```

Parameters

1. Number
The sequence number of the coin or note acceptor being configured.
2. Snapshot
A pointer to a program buffer containing the configuration data for the specified acceptor. See below for details.

Return Value

None.

Remarks

The sequence numbers of the acceptors are contiguous and run from zero upwards.

A call to **ReadAcceptorDetails** followed by call to **WriteAcceptorDetails** for the same data will have no effect on the system.

ReadDispenserDetails

Synopsis

The **ReadDispenserDetails** call provides a snapshot of all the information possessed by the interface on a single unit of money dispensing equipment.

```
int ReadDispenserDetails( int           Number,  
                          DispenserBlock* Snapshot) ;
```

Parameters

1. Number
The sequence number of the coin or note dispenser about which information is required.
2. Snapshot
A pointer to a program buffer, into which all of the information about the specified dispenser will be copied.

Return Value

Non zero if the specified input device exists, Zero if the end of the list is reached.

Remarks

The sequence numbers of the dispensers are contiguous and run from zero upwards.

WriteDispenserDetails

Synopsis

The **WriteDispenserDetails** call updates all the changeable information to the interface for a single unit of money handling equipment.

```
void WriteDispenserDetails( int           Number,  
                           DispenserBlock* Snapshot) ;
```

Parameters

1. Number
The sequence number of the coin or note dispenser being configured.
2. Snapshot
A pointer to a program buffer containing the configuration data for the specified dispenser. See below for details.

Return Value

None.

Remarks

The sequence numbers of the dispensers are contiguous and run from zero upwards. A call to **ReadDispenserDetails** followed by call to **WriteDispenserDetails** for the same data will have no effect on the system.

Dispenser Value Reassignment

Where a coin dispenser does not have an inbuilt value, Paylink allows the value of the coin associated with a Dispenser to be re-assigned. (This can override the value defined in the configuration file.) This is ignored for **all** recyclers

To do this:

- the dispenser to be updated should be found using **ReadDispenserDetails()**,
- the **Dispenser.Value** updated to the new value,
- set the **Dispenser.Status** field to DISPENSER_REASSIGN_VALUE
- and **WriteDispenserDetails()** used to update the record to Paylink.
-

Paylink will acknowledge that the update has been processed by updating the **Dispenser.Status** field returned by a subsequent **ReadDispenserDetails()** to DISPENSER_VALUE_REASSIGNED. *If this value is not seen in the **Dispenser.Status** field, then the value change has not be processed by Paylink.*

Dispenser Dump

All bill recyclers support a facility whereby a Dispenser can be dumped to the cash box. Paylink supports this facility by use of writing 2 special values in the Status field. To use this to dump all the bills / notes to the cashbox:

- the dispenser to be updated should be found using **ReadDispenserDetails()**,
- set the **Dispenser.Status** field to DISPENSER_CASHBOX_DUMP
- and call **WriteDispenserDetails()** to update the record to Paylink.

To use this to dump a limited number of bills / notes to the cashbox:

- the dispenser to be updated should be found using **ReadDispenserDetails()**,
- set the count of bills that are to be dumped in the **Dispenser.NotesToDump** field
- set the **Dispenser.Status** field to DISPENSER_PARTIAL_DUMP
- and call **WriteDispenserDetails()** to update the record to Paylink.

The dump will proceed, and progress can usually be seen by the **DispenserBlock.CoinCount** field returned by a subsequent **ReadDispenserDetails()** decreasing. When the dump processing completes the return value of **Dispenser.Status** will change to DISPENSER_DUMP_FINISHED if the dump completed normally, or to an error status if there was a problem.

Single Note Head Escrow

Escrow Overview

All note acceptors provide a facility known as “escrow”, whereby after the note has been identified it is held within the acceptor and can then be either returned to the user or fully accepted and stacked (if a stacker is in use).

Paylink *always* uses this facility (unless it is not available) as it provides enhanced security during the note acceptance process. (As Paylink issues an accept for each note from escrow it therefore *has* to have accurately accounted for how many it has read.)

By *default*, the Paylink system automatically issues an accept command as soon as note is reported in the escrow position.

If the application wishes to have more control over the acceptance process, Paylink provides “note in escrow” facilities that allow the Application full control over whether to issue an accept command or to issue a reject command.

Escrow system usage

The Paylink unit fully supports escrow system usage. It reports the note that is currently held in escrow by an acceptor, and allows the application to either return or accept the escrow holding of the acceptor.

In most system only one escrow capable acceptor will be present, the Paylink unit will however support escrow on an unlimited number of acceptors. In order to allow for accurate information and control to pass between the application and the Paylink firmware, the escrow holding reported is limited to a single acceptor at time. If two acceptors are holding escrow at the same time, the second will not be reported until the first has completed.

At start-up, the system does not report escrow details and all acceptors are run in “normal” mode where all currency is accepted. For the application to use escrow, the call **EscrowEnable** is issued. Following this the call **EscrowThroughput** will return the *total* value of all currency that has ever been held in escrow (in the same way as for **CurrentValue**, except that the value is not preserved over resets). An increase in the value returned indicates that a note is now in escrow. The **HeldInEscrow** field within the **AcceptorCoin** structure will indicate the number of each note / coin that is currently being held.

The **EscrowAccept** call will cause the Paylink unit to complete the acceptance of the currency in question. When complete, this will be indicated by an increase in **CurrentValue**. An **EscrowReturn** call will cause the currency to be returned with no further indication to the application. Following either call, the **EscrowThroughput** value may increase immediately due to another acceptor having an escrow holding.

If the application wishes to stop using the escrow facilities, it may issue the **EscrowDisable** call. This will have the side effect of accepting any outstanding escrow holds.

EscrowEnable

Synopsis

Change the mode of operation of all escrow capable acceptors to hold inserted currency in escrow until a call of **EscrowAccept**.

The **EscrowEnable** call is used to start using the escrow system

```
void EscrowEnable (void) ;
```

Parameters

None

Return Value

None

EscrowDisable

Synopsis

Change the mode of operation of all escrow capable acceptors back to the default mode in which all currency is fully accepted on insertion

```
void EscrowDisable (void) ;
```

Parameters

None

Return Value

None

Remarks

1. If any currency is currently held in escrow when this call is made, it will be accepted without comment.

EscrowThroughput

Synopsis

Determine the cumulative monetary value that has been held in escrow since the system was reset.

The **EscrowThroughput** call is used to determine the cumulative total value of all coins and notes read by the money handling equipment that have ever been held in escrow.

int EscrowThroughput (void) ;

Parameters

None

Return Value

The current value, in the lowest denomination of the currency (i.e. cents / pence etc.) of all coins and notes ever held in Escrow.

Remarks

1. It is the responsibility of the application to keep track of value that has been accepted and to monitor for new coin / note insertions by increases in the returned value.
2. Note that this value should be read following the call to **OpenMHE** and before the call to **EnableInterface / EscrowEnable** to establish a starting point before any coins or notes are read.
3. If the acceptor auto-returns the coin / note then this will fall to its previous value. This can (potentially) occur *after* a call to **EscrowAccept()** or **EscrowReturn()** if the acceptor has already started its return sequence.

EscrowAccept

Synopsis

If the acceptor that was last reported as holding currency in escrow is still in that state, this call will cause it to accept that currency.

void EscrowAccept (void) ;

Parameters

None

Return Value

None

Remarks

1. If a second acceptor has (unreported) currency in escrow at the time this call is made, it will immediately cause the **EscrowThroughput** to be updated.
2. If no currency is currently held in escrow when this call is made, it will be silently ignored.

EscrowReturn

Synopsis

If the acceptor that was last reported as holding currency in escrow is still in that state, this call will cause it to return that currency.

void EscrowReturn (void) ;

Parameters

None

Return Value

None

Remarks

1. If a second acceptor has (unreported) currency in escrow at the time this call is made, it will immediately cause the **EscrowThroughput** to be updated.
2. If no currency is currently held in escrow when this call is made, it will be silently ignored.

Multi note Extended Escrow (1.12.6)

Overview

The pre-requisite for Extended Escrow is that the note recycler has firmware installed so that at least one recycling unit, called the escrow recycling unit in this document, that will accept every denomination of note and that has ability to either transfer these notes to the acceptor's stacker / cash box, or to return them to the user. Note that this is a requirement on the *recycler* firmware.

The essential feature of the Paylink Extended Escrow system is that notes are initially accepted into a logical escrow, with the Paylink unit keeping track of which notes have been inserted and where they are being stored.

The application, which uses the Paylink escrow facilities to control and monitor this, can then decide to either stack (keep) the notes, or can decide to return the notes to the user.

The code running within Paylink is responsible for tracking the notes and issuing the appropriate commands to the note acceptor.

Accepting Notes

When the `EXT_ESCROW_ACCEPT` command is issued, notes are accepted to the recycling unit(s) and the `EscrowNote` array is filled in, to detail which notes have been accepted and which recycling unit they are being stored on.

Returning Notes

If an `EXT_ESCROW_RETURN` command is issued, then for every recycling unit Paylink issues a "pay" command to the recycler for the number of notes stored on that unit during the accept phase - thereby returning to the user the notes they have just inserted.

Keeping Notes

If an `EXT_ESCROW_STACK` command is issued, then the application wishes to keep the notes, and the processing varies depending upon which notes have been stored and which recycling units they are store on.

For an escrow recycling unit that has been configured to be "pure escrow" the Paylink code issues a dump command to transfer all the notes in that unit to the cashbox.

EscrowControlBlock

Most of the feedback from Paylink to the API takes place using a single EscrowControlBlock, which in turn contains an array of EscrowNoteDetails. These have the following layout:

Fields / properties for the EscrowControlBlock Object

Field Type	Name	Meaning
int	EscrowVersion	The version of the escrow system available.
int	State	The current state of the Escrow System
int	Result	The result of the previous Escrow Command
int	TotalValue	The total values of all the notes in Escrow
int	ValueReturned	The total values of all the notes just returned from Escrow
int	AcceptorNo	The index of the acceptor running this escrow
int	NoInEscrow	The number of notes currently in escrow
EscrowNoteDetails	EscrowNote[MAX_ESCROW]	Details on the notes in escrow

Fields / properties for the EscrowNoteDetails sub object

Field Type	Name	Meaning
int	Value	Value of this note
int	NoteNumber	The index of the AcceptorCoin in the AcceptorBlock
int	Location	The dispenser on which this note is stored
int	Status	Set if this note could not be returned / stacked.

Constants for Escrow State

Name	Meaning
EXT_ESCROW_NONE	This Paylink system does not implement extended escrow
EXT_ESCROW_OFF	The extended escrow system is not running.
EXT_ESCROW_IDLE	The extended escrow system is running, but not doing anything
EXT_ESCROW_WAITING	The extended escrow system is running, waiting for notes, but empty
EXT_ESCROW_LOADING	Notes are in transit within the escrow system
EXT_ESCROW_STORED	The escrow system is idle, storing notes.
EXT_ESCROW_PAUSED	The escrow system is storing notes, but not accepting any more
EXT_ESCROW_STACKING	The escrow system is transferring notes to the cash box.
EXT_ESCROW_RETURNING	The escrow system is returning notes to the user.
EXT_ESCROW_RETURNED_OK	The escrow system has finished returning notes to the user
EXT_ESCROW_RETURNING_PROBLEM	While returning notes to the user, there was a problem
EXT_ESCROW_STACKED_OK	The escrow system has finished transferring notes to the cash box.
EXT_ESCROW_STACKING_PROBLEM	The process of stacking notes failed.
EXT_ESCROW_FULL	The escrow system is full, the acceptor is disabled
EXT_ESCROW_POWER_ACTIVE	The escrow system was running when Paylink was powered off

Constants for Escrow Result

Name	Meaning
EXT_ESCROW_COMPLETE	The last escrow command completed correctly, the Escrow block now reflects that.
EXT_ESCROW_BUSY	An escrow command has been accepted, but not completed. The Escrow block is invalid.
EXT_ESCROW_WRONGSTATE	A valid escrow command was issued, but the Escrow system was unable to action it.
EXT_ESCROW_ERROR	An invalid escrow command was issued.

Constants for Escrow Command

Name	Meaning
EXT_ESCROW_START	Turn on the Escrow system
EXT_ESCROW_STOP	Turn off the Escrow system
EXT_ESCROW_ACCEPT	Allows the acceptor to input notes to the Escrow system
EXT_ESCROW_PAUSE	Stop accepting notes and allow system to settle.
EXT_ESCROW_STACK	Transfer notes to the cash box (or retain them on Rolls)
EXT_ESCROW_RETURN	Return all escrowed notes to the user.

Constants for Status in EscrowNoteDetails

Name	Meaning
ESCROW_NOTE_HELD	The note is currently being held in Escrow
ESCROW_NOTE_STACKED	This note was Stacked to the cashbox
ESCROW_NOTE_RETURNED	This note was returned
ESCROW_NOTE_PROBLEM	This note could not be returned / stacked

Operation

The operation of the Extended Escrow is quite simple. The Application can at all times discover the state of the escrow system by calling **ReadEscrowBlock**.

The current state of the system is reported in the **State** field.

To control the Escrow system, the application should call **EscrowCommand** with an appropriate parameter. The Escrow system indicates that it has accepted (and is processing) the command by setting the **Result** field to **EXT_ESCROW_COMPLETE**.

For each Escrow system state, the acceptable commands and their results are shown in this table: (for ease of layout, the EXT_ESCROW_ prefix for all the states / commands has been omitted)

Current State	Allowable Commands / Events	New State	Comments
NONE	N/A		No escrow system configured
OFF	START	IDLE	
	START	RETURNED_PROBLEM	If there were notes stored
IDLE	ACCEPT	WAITING	No notes have yet been read
	STOP	OFF	
WAITING	Note being read	LOADING	
	PAUSE	IDLE	
LOADING	Read completed	STORED	This repeats for each note
STORED	Note being read	LOADING	
	Note read that fills the system	FULL	At this point, the acceptor is disabled.
	PAUSE	PAUSED	At this point, the acceptor is disabled.
PAUSED	ACCEPT	LOADING	Acceptance is restarted
	STACK	STACKING	
	RETURN	RETURNING	
STACKING	Stacking OK	STACKED_OK	
	Stacking problem	STACKED_PROBLEM	
RETURNING	Returning OK	RETURNED_OK	
	Returning problem	RETURNED_PROBLEM	
STACKED_OK	ACCEPT	WAITING	This clears the previous transaction
	STOP	OFF	
STACKING_PROBLEM	Problem fixed	STACKED_OK	This status stays until fixed
RETURNED_OK	ACCEPT	WAITING	This clears the previous transaction
	STOP	OFF	
RETURNING_PROBLEM	RETURN	RETURNING	Retry the return
FULL	PAUSE	PAUSED	

Abnormal Situations

If there are notes in the extended escrow dispenser at start up, they need to be returned to the user. To allow the application control over the timing of this return, the extended escrow system transitions from `OFF` to `RETURNED_PROBLEM` when the `START` command is issued.

ReadEscrowBlock

Synopsis

The **ReadEscrowBlock** call is used to obtain the latest information for an EscrowControlBlock.

```
int ReadEscrowBlock (int                Number,  
                    EscrowControlBlock* Snapshot) ;
```

Parameters

3. Number
The sequence number of the escrow control system about which information is required.
4. Snapshot
A pointer to a program buffer into which all the information about the specified acceptor will be copied.

Return Value

Non zero if the specified Escrow control block exists, Zero if the end of the list is reached.

Remarks

3. Zero can be returned when Number has the value of zero if no escrow control systems are running.

EscrowCommand

Synopsis

The EscrowCommand call is used by the application to handle all interaction with the extended escrow system.

```
void EscrowCommand (int      Number  
                    int      Command) ;
```

Parameters

1. Number
The sequence number of the escrow control system for which the command is intended.
2. Command
The command being issued..

Return Value

None

Remarks

1. The success or failure and subsequent progress of a command are determined by value set into the **Result** and **State** fields of the EscrowControlBlock.
2. Immediately following this call, the **Result** field of the EscrowControlBlock will *a/ways* be `EXT_ESCROW_BUSY`.
3. For any particular **State** of the escrow control system only a small subset of commands are valid. Any other command will generate `EXT_ESCROW_WRONGSTATE`.

Bar Codes

Where an acceptor provides barcode facilities, the Paylink unit can fully support this by enabling bar code acceptance and reporting the barcodes read.

Barcode reading is always handled using the escrow position on the acceptor, in a similar way to that already described. The barcode is held in the acceptor pending a call from the application that will either stack or return it.

In most systems, only one barcode capable acceptor will be present, the Paylink unit will however support barcodes on an unlimited number of acceptors. In order to allow for accurate information and control to pass between the application and the Paylink firmware, the barcode reported is limited to a single acceptor at time. If two acceptors are holding barcoded tickets at the same time, the second will not be reported until the first has completed.

The basic barcodes processed by the IMHEI system are in the format "Interleaved 2 of 5" and are 18 characters long. The basic functions therefore return a 19 character NULL terminated string.

Later barcode systems now return up to 40 characters, so API functions with the suffix Ext have been added, which will handle any length of barcode.

Basic Barcodes can also be printed if a dedicated barcode printer is connected.

BarcodeEnable

Synopsis

Change the mode of operation of all Barcode capable acceptors to accept tickets with barcodes on them.

The **BarcodeEnable** call is used to start using the Barcode system

```
void BarcodeEnable (void) ;
```

Parameters

None

Return Value

None

BarcodeDisable

Synopsis

Change the mode of operation of all Barcode capable acceptors back to the default mode in which only currency is accepted.

```
void BarcodeDisable (void) ;
```

Parameters

None

Return Value

None

Remarks

1. If a Barcoded ticket is currently held when this call is made, it will be returned without comment.

BarcodeInEscrow / BarcodeInEscrowExt

Synopsis

This is the regular “polling” call that the application should make into the DLL to obtain the current status of the barcode system. If a barcode is read by an acceptor, it will be held in escrow and this call will return true in notification of the fact.

Originally, barcodes from note acceptors were always 18 characters plus a trailing NULL, recently the size of the barcode has increase, so a new function is provided to handle these larger barcodes.

C++

```
int BarcodeInEscrow (char BarcodeString[19]) ;  
int BarcodeInEscrowExt (char* BarcodeString,  
                        int    BufferLength) ;
```

C#, VB

```
int BarcodeInEscrow (ref string BarcodeString) ;  
int BarcodeInEscrowExt (ref string BarcodeString,  
                        int    BufferLength) ;
```

Java

```
int BarcodeInEscrow (string BarcodeString[1]) ;  
int BarcodeInEscrowExt (string BarcodeString[1],  
                        int    BufferLength) ;
```

Parameters

1. BarcodeString
A pointer to a string / buffer of at least 19 (or BufferLength) characters into which the last barcode read from any acceptor is placed. This will be all NULL if no barcoded ticket has been read since system start-up.
2. BufferLength
The length of the provided buffer.

Return Value

The return value is non zero if there is a barcode ticket currently held in an Acceptor, zero if there is not.

Remarks

1. There is no guarantee that at the time the call is made the acceptor has not irrevocably decided to auto-eject the ticket.
2. Whilst this function is returning no zero, then *exactly one* of the Acceptors accessed by ReadAcceptorDetails will have an EscrowBarcodeHere field with a non-zero value.

BarcodeStacked / BarcodeStackedExt

Synopsis

Following a call to **BarcodeAccept** the system *may* complete the reading of a barcoded ticket. If it does, then the count returned by **BarcodeStacked** will increment. There is no guarantee that this will take place, so the application should continue to poll **BarcodeInEscrow**.

Originally, barcodes from note acceptors were always 18 characters plus a trailing NULL, recently the size of the barcode has increase, so a new function is provided to handle these larger barcodes.

C++

```
int BarcodeStacked (char BarcodeString[19]) ;  
int BarcodeStackedExt (char BarcodeString[19],  
                      int BufferLength) ;
```

C#, VB

```
int BarcodeStacked (ref string BarcodeString) ;  
int BarcodeStackedExt (ref string BarcodeString,  
                      int BufferLength) ;
```

Java

```
int BarcodeStacked (string BarcodeString[1]) ;  
int BarcodeStackedExt (string BarcodeString[1],  
                      int BufferLength) ;
```

Parameters

1. BarcodeString
A pointer to a string / buffer of at least 19 (or BufferLength) characters into which the last barcode read from any acceptor is placed. This will be all NULL if no barcoded ticket has been read since system start-up.
2. BufferLength
The length of the provided buffer.

Return Value

The count of all the barcoded tickets that have been stacked since system start-up. An increase in this value indicates that the current ticket has been stacked - its contents will be in the **BarcodeString** buffer.

Remarks

1. It is the responsibility of the application to keep track of the number of tickets that have been accepted and to monitor for new insertions by increases in the returned value.
2. Note that this value should be read following the call to **OpenMHE** and before the call to **EnableInterface / BarcodeEnable** to establish a starting point before any new tickets are read.
3. Whenever the value returned by this is incremented, then *exactly one* of the Acceptors accessed by ReadAcceptorDetails will have a BarcodesStacked field that also increments.

BarcodeAccept

Synopsis

If the acceptor that was last reported as holding a Barcode ticket is still in that state, this call will cause it to accept that ticket.

void BarcodeAccept (void) ;

Parameters

None

Return Value

None

Remarks

1. If a second acceptor has (unreported) currency in Barcode at the time this call is made, it will immediately cause the **BarcodeTicket** to be updated.
2. If no ticket is currently held when this call is made, it will be silently ignored.

BarcodeReturn

Synopsis

If the acceptor that was last reported as holding a Barcode ticket is still in that state, this call will cause it to return that ticket.

void BarcodeReturn (void) ;

Parameters

None

Return Value

None

Remarks

1. If a second acceptor has (unreported) currency in Barcode at the time this call is made, it will immediately cause the **BarcodeTicket** to be updated.
2. If no ticket is currently held when this call is made, it will be silently ignored.

Barcode Printing

BarcodePrint

Synopsis

This call is used to print a barcoded ticket, if the Paylink system supports a printer.

```
void BarcodePrint (TicketDescription* TicketContents) ;
```

Parameters

1. TicketContents.
Pointer to a TicketDescription structure that holds pointers to the strings that the application is "filling in". NULL pointers will cause the relevant fields to default (usually to blanks).

Fields / properties for the TicketDescription Object / Structure

Field Type	Name	Comment
int	TicketType	The "template" for the ticket
char* / string	BarcodeData	
char* / string	AmountInWords	
char* / string	AmountAsNumber	But still a string
char* / string	MachineIdentity	
char* / string	DatePrinted	
char* / string	TimePrinted	

Return Value

None

Remarks

1. There are a number of fields that can be printed a barcode ticket.
Rather than provide a function with a large number of possibly null parameters, we use a structure, which may have fields added to end.
The user should ensure that all unused pointers are zero.
2. Before issuing this call the application should ensure that **BarcodePrintStatus** has returned a status of **PRINTER_IDLE**
3. The mechanics of the printing mechanism rely on **BarcodePrintStatus** being called regularly after this call, in order to "stage" the data to the interface.

BarcodePrintStatus

Synopsis

This call is used to determine the status of the barcoded ticket printing system.

```
int BarcodePrintStatus (void) ;
```

Parameters

None

Return Value

Mnemonic	Value	Meaning
PRINTER_NONE	0	Printer completely non functional / not present
PRINTER_FAULT	0x80000000	There is a fault somewhere
PRINTER_IDLE	0x00000001	The printer is OK / Idle /Finished
PRINTER_BUSY	0x00000002	Printing is currently taking place
PRINTER_PLATEN_UP	0x00000004	
PRINTER_PAPER_OUT	0x00000008	
PRINTER_HEAD_FAULT	0x00000010	
PRINTER_VOLT_FAULT	0x00000040	
PRINTER_TEMP_FAULT	0x00000080	
PRINTER_INTERNAL_ERROR	0x00000100	
PRINTER_PAPER_IN_CHUTE	0x00000200	
PRINTER_OFFLINE	0x00000400	
PRINTER_MISSING_SUPPY_INDEX	0x00000800	
PRINTER_CUTTER_FAULT	0x00001000	
PRINTER_PAPER_JAM	0x00002000	
PRINTER_PAPER_LOW	0x00004000	
PRINTER_NOT_TOP_OF_FORM	0x00008000	
PRINTER_OPEN	0x00010000	
PRINTER_TOP_OF_FORM	0x00020000	
PRINTER_JUST_RESET	0x00040000	

Remarks

1. The mechanics of the printing mechanism rely on this being called regularly after the **BarcodePrint** call, in order to “stage” the data to the interface, until **PRINTER_BUSY** is no longer returned.
2. Any reported fault that requires an operator action will cause the **PRINTER_FAULT** bit to be set.
3. A **PRINTER_NONE** status will be reported if the printer is powered off after having been working.

Cashless Processing

Background

There are a number of cashless systems available that are supported by Paylink. These include: Credit Card Acceptance and Ticket In / Out applications for AWP systems.

Paylink provides an interface to support this for acquiring credit (from a ticket or from a credit card).

In all cases, the remote processing runs autonomously, and Paylink provides a structure that reflects the processing of the remote unit, which should be regularly polled by application in order to determine what is happening.

The ways in which this object and functions interact to complete cashless transactions is described in the Cashless Processing section of the "Milan / Paylink System Manual".

Note that MaximumPay is still used even when this information is not available from the remote system. It operates as a flag; if it is non zero then credit payment is possible.

Fields / properties for the Cashless Object

Field Type	Name	Meaning
Static Information		
int	CashlessType	See below
char*	Description	Cashless device details
int	SerialNumber	
int	TotalAcquisitions	The lifetime total count of cashless credit acquisition..
int	TotalCredit	The lifetime total of cashless credit that has been acquired.
Dynamic Transaction Information		
int	CurrentState	The state of the system / current transaction - See below
int	StateDetails	Extra Details for the current state
int	CreditValue	The amount of credit from Paylink for the current status
char*	ReferenceData	Optional current ticket / card reference number.
int	TotalPaid	The lifetime total of credit successfully transferred to the remote system
int	MaximumPay	The maximum value of any requested payment

Values for the CashlessType item.

Name	Value	
CR_CREDIT_CARD	0	This is a variable credit card based system
CR_TICKET	1	This is a Fixed credit ticket based system

Values for the CurrentState item.

Name	Value	
CR_NO_UNIT	0	No appropriate unit connected
CR_BUSY	1	Busy
CR_DISABLED	2	Idle and disabled
CR_IDLE	3	Idle
CR_FAULT_DISABLED	4	The device has become non operational
CR_FAULT_IDLE	5	The device has become non operational
CR_AVAILABLE_IDLE	6	Idle, but prepared for Credit Requests
CR_INVALID_OP	100	Invalid operation
These are only valid during Credit Input operations		
CR_AVAILABLE	11	Arbitrary Credit Available
CR_CONFIRMED	12	Credit Request from Application Accepted
CR_TAKEN	13	Credit Actually Taken for remote source
CR_REFUSED	14	Credit Request from Application Refused
CR_CANCELLED	15	Application has successfully cancelled the transaction
CR_FAILED	16	The device failed during processing - no credit taken
CR_FAILED_TAKEN	17	The device failed during processing - credit was taken
CR_INVALID_REF	101	Invalid Reference
These are only valid during Credit Payment operations		
CR_TRANSFERRING	21	A credit output operation is in progress
CR_TRANSFERED	22	The Credit has been accepted by the device
CR_TRANSFER_FAIL	23	The credit has been refused by the device

Abnormal Processing

A number of abnormal situations can occur, especially if part of the system is reset during processing. To aid recovery of these situations, three lifetime counters are provided: TotalAcquisitions, TotalCredit and Total Paid, if these are saved whenever the application is idle, then the possibly updated values can aid in determining how far an interrupted transaction got.

CashlessReadData

Synopsis

Return the current information related to the delivery of credit from the cashless system.

```
int CashlessReadData(int Number,  

                    Cashless* DataBlock) ;
```

Parameters

1. Number
The sequence number of the Cashless peripheral
2. DataBlock
A pointer to a Cashless structure that is to be updated.

Return Value

Zero if the sequence number refers to a non-existent peripheral. The CurrentState item in the updated DataBlock will have the value CR_NO_UNIT.

Remarks

1. This function should be called regularly to monitor the Cashless facility.

CashlessEnable

Synopsis

Where relevant (e.g. card systems) this enables peripheral equipment to be start accepting requests. This is also used following any transaction sequence, to return the system to state of CR_IDLE.

```
void CashlessEnable (int Number) ;
```

Parameters

5. Number
The sequence number of the Cashless peripheral

Return Value

None

Remarks

2. With ticket based systems this has no effect on the peripheral, as the application is responsible for initiating the process
3. Following this call, CurrentState should become CR_IDLE, CR_AVAILABLE_IDLE or CR_FAULT_IDLE

CashlessDisable

Synopsis

Where relevant (e.g. card systems) this causes peripheral equipment to cease accepting requests.

```
void CashlessDisable (int Number) ;
```

Parameters

1. Number
The sequence number of the Cashless peripheral

Return Value

None

Remarks

1. With ticket based systems this has no effect on the peripheral, as the application is responsible for initiating the process
2. Following this call, CurrentState should become CR_DISABLED (or CR_FAULT_DISABLED)

SubmitTicket

Synopsis

Only relevant for ticket based system, this provides the reference number read from a barcoded ticket, in the expectation of this resulting in a credit becoming available.

```
void SubmitTicket(int Number,  
                  char* TicketReference) ;
```

Parameters

1. Number
The sequence number of the Cashless peripheral
2. TicketReference
A pointer to a ASCII string containing the reference number.

Return Value

N/A

Remarks

1. This function should cause CurrentState to eventually become CR_AVAILABLE, with the value of the ticket shown in the CreditDelivered.

CashlessRequestCredit

Synopsis

When Paylink is indicating that credit is available, this call allows the application to notify Paylink how much of the available credit the application wishes to take.

```
void CashlessRequestCredit(int Number,  
                           int AmountRequested) ;
```

Parameters

1. Number
The sequence number of the Cashless peripheral
2. AmountRequested
The amount of credit that the application is requesting from the source of credit.

Return Value

N/A

Remarks

1. This must only be called when CurrentState is CR_AVAILABLE or is CR_AVAILABLE_IDLE.
2. Some peripheral configurations (typically ticket based) will only allow the AmountRequested to be equal to CreditValue and for CR_AVAILABLE_IDLE the value is only that used in Standard.cfg, not part of the cashless system.
3. The credit described may not actually be available. The application must wait for a state of CR_CONFIRMED before starting to use any credit.
4. Under error conditions; following this call, the state of CR_FAILED_TAKEN may be reached, even without the application calling CashlessTakeCredit().
5. If the application doesn't want to use the available credit, it should call **CashlessRefuseCredit** instead.

CashlessRefuseCredit

Synopsis

When Paylink is indicating that credit is available, this call allows the application to notify Paylink that the application wishes to cancel the transaction.

```
void CashlessRefuseCredit(int Number) ;
```

Parameters

1. Number
The sequence number of the Cashless peripheral.

Return Value

N/A

Remarks

1. This must only be called when CurrentState is CR_AVAILABLE.
 2. This call will be complete when CurrentState is CR_REFUSED.
-

CashlessTakeCredit

Synopsis

When Paylink is indicating that credit has been confirmed, this is the final call to commit the transaction.

```
void CashlessTakeCredit(int Number) ;
```

Parameters

1. Number
The sequence number of the Cashless peripheral

Return Value

N/A

Remarks

1. This must only be called when CurrentState is CR_CONFIRMED
2. The application must wait for the subsequent state of CR_TAKEN before regarding the transaction as having completed.

CashlessCancelCredit

Synopsis

When Paylink is indicating that credit has been confirmed with CR_CONFIRMED, this is a "last chance" to cancel the transaction because there has been a problem using the credit. This call may or may not succeed.

```
void CashlessCancelCredit(int Number) ;
```

Parameters

1. Number
The sequence number of the Cashless peripheral

Return Value

N/A

Remarks

1. This must only be called when CurrentState is CR_CONFIRMED
2. *If successful*, CurrentState will become CR_CANCELLED
3. The application **may** find a subsequent state of CR_FAILED_TAKEN and will then have to take an appropriate alternative action for the credit specified in CreditValue.

CashlessPayCredit

Synopsis

This must be called when CurrentState is CR_IDLE or CR_AVAILABLE_IDLE. Paylink will attempt to transfer the amount specified to the cashless peripheral. Following the call the value returned in the CurrentState be CR_TRANSFERRING until the payment terminates.

```
void CashlessPayCredit(int Number) ,  
                        int CreditAmount) ;
```

Parameters

1. Number
The sequence number of the Cashless peripheral
2. CreditAmount
The value to be transferred to the external system

Return Value

N/A

Remarks

1. This must only be called when CurrentState is CR_AVAILABLE or is CR_AVAILABLE_IDLE.
2. *If successful*:
 - CurrentState will eventually become CR_TRANSFERRED
 - TotalPaid will increment by CreditAmount
3. *If the credit is not successfully transferred*.
 - CurrentState will eventually become CR_FAILED
 - TotalPaid will remain unchanged
4. The value of TotalPaid is retained over a power cycle.

CashlessReset

Synopsis

This call notifies the Paylink cashless system that the application is now ready to start another cashless transaction.

void CashlessReset(int Number) ;

Parameters

1. Number
The sequence number of the Cashless peripheral

Return Value

N/A

Remarks

1. Following this call the cashless system is guaranteed to return to CR_IDLE (or CR_FAULT_IDLE).

Meters / Counters

The Paylink units support the concept of external meters that are accessible from the outside of the PC system.

In keeping with the Paylink concept, an interface is defined to an idealised meter. This will be implemented transparently by the card using the available hardware. Currently the Paylink unit will support either a **Starpoint Electronic Counter**, or from 1 to 8 mechanical meters.

Mechanical Meters (1.12.4)

From 1.12.4 onwards, Paylink supports mechanical meters, driven using pulses through the general-purpose high power outputs. Suitable meters are required to operate on DC at a speed of at least 20 pulses per second.

Configuration file entries are used to map Counter Numbers 1 to 8 onto the Paylink outputs.

Paylink records how many pulses have been sent, and how many are currently required. It attempts to handle the fact that while the pulses are being output the power may be cycled. Paylink updates its non-volatile memory as it turns on the transistor at the start of the pulse. This means that during a power cycle at most one pulse may be lost (as it is not driven for long enough) but no spurious pulses can be generated.

CounterIncrement

Synopsis

The **CounterIncrement** call is made by the application to increment a specific counter value.

```
void CounterIncrement(int CounterNo,  
                      int Increment);
```

Parameters

1. CounterNo
This is the number of the counter to be incremented.
2. Increment
This is the value to be added to the specified counter.

Return Value

None

Remarks

1. If the counter specified is higher than the highest supported by the current hardware, then the call is silently ignored.

CounterCaption

Synopsis

The **CounterCaption** call is used to associate a caption with the specified counter. This is related to the **CounterDisplay** call described below.

C++

```
void CounterCaption(int CounterNo,  
                    char* Caption);
```

C#, VB, Java

```
void CounterCaption(int CounterNo,  
                    string Caption);
```

```
void CounterCaption(long CounterNo,
```

Parameters

1. CounterNo
This is the number of the counter to be associated with the caption.
2. Caption
This is an ASCII string that will be associated with the counter.

Return Value

None

Remarks

1. The meter hardware may have limited display capability. It is the system designer's responsibility to use captions that are within the meter hardware's capabilities.
2. If the counter specified is higher than the highest supported, then the call is silently ignored.
3. The specified caption is **not** stored in the meter, even if the meter offers this facility.
4. This is not relevant for mechanical meters.

CounterRead

Synopsis

The **CounterRead** call is made by the application to obtain a specific counter value as stored by the meter interface.

```
int CounterRead(int CounterNo) ;
```

Parameters

1. CounterNo
This is the number of the counter to be incremented.

Return Value

The Value of the specified meter read at system start-up and incremented by subsequent **CounterIncrement** calls.

Remarks

1. If the counter specified is higher than the highest supported, then the call returns -1
2. If error conditions have prevented an electronic meter updating, this call will show the value it **should** be at, **not** the actual value. (The value is read only read from the meter at system start-up.)
3. For a mechanical meter, the total of all increment calls made to Paylink is stored and returned by this call.

ReadCounterCaption

Synopsis

The **ReadCounterCaption** call is used to determine the caption for the specified counter

C++

```
char* CounterCaption(int CounterNo) ;
```

C#, VB, Java

```
string CounterCaption(int CounterNo) ;
```

Parameters

1. CounterNo
This is the number of the counter to be incremented.

Return Value

None

Remarks

1. If the counter specified is higher than the highest supported, then the call returns an empty string ("").
2. If available, captions stored in the meter are read out at system start-up and used to initialise the captions used by the interface.

CounterDisplay

Synopsis

The **CounterDisplay** call is used to control what is displayed on an electronic meter.

```
void CounterDisplay (int DisplayCode) ;
```

Parameters

4. DisplayCode

If positive, this specifies the counter that will be continuously display by the meter hardware.

If negative, then the display will cycle between the caption (if set) for 1 second followed by the value for 2 seconds, for the counter corresponding to the positive form of the code

Return Value

None

Remarks

4. This result of this call with a negative parameter is undefined if the counter has an associated caption.
5. Whenever the meter displayed is changed, the caption (if set) is unconditionally displayed for one second.
6. This is not relevant for mechanical meters.

MeterStatus

Synopsis

The **MeterStatus** call is used determine whether working meter equipment is connected.

```
int MeterStatus (void) ;
```

Parameters

None

Return Value

One of the following:

Value	Meaning	Mnemonic
0	A Meter is present and working correctly	METER_OK
1	No Meter has ever been found	METER_MISSING
2	The Meter is no longer functioning	METER_DIED
3	The Meter is functioning, but is itself reporting internal problems	METER_FAILED

Remarks

1. For Mechanical Meters, METER_OK is returned for correctly defined outputs. Paylink has no way of detecting whether anything is actually connected to the outputs.

MeterSerialNo

Synopsis

The **MeterSerialNo** call is used determine which item meter equipment is connected.

```
int MeterSerialNo ( void );
```

Parameters

None

Return Value

The 32-bit serial number retrieved from the meter equipment.

Remarks

1. Where the meter equipment is not present or does not have serial number capabilities, zero is returned.

E²Prom

Included in the Paylink unit is E²PROM memory, which is used by the embedded process to maintain counters etc. 256 bytes of this E²PROM is available to users to store essential information if they wish to run their system with no other writeable storage.

In this section, routines are described to access this user storage and to allow for a user application to clear all the E²PROM memory on the card, after testing and before delivery to an end user.

E2PromReset

Synopsis

The **E2PromReset** call is made by the application to clear all the *internal* E²PROM memory on the card. This is where the Paylink system keeps the value in / value out counters, the configuration information, etc.

```
void E2PromReset(int LockE2Prom) ;
```

Parameters

1. LockE2Prom
This is a flag. If zero, then the E2PROM may be reset again later.
If non zero, then **all** future calls to this function will have no effect on the card.

Return Value

None

Remarks

An example application for this is available within the SDK folder structure.

E2PromWrite

Synopsis

The **E2PromWrite** call is made by the application to write to all or part of the user E²PROM on the card.

C++

```
void E2PromWrite (void* UserBuffer,  
                  int   BufferLength) ;
```

C#, VB, Java

```
void E2PromWrite (Byte[] UserBuffer,  
                  int     BufferLength) ;
```

Parameters

1. **UserBuffer**
This is the address of the user's buffer, from which **BufferLength** bytes of data are copied to the start of the user area.
2. **BufferLength**
This is the count of the number bytes to be transferred. If this is greater than 256 the extra will be silently ignored.

Return Value

None

Remarks

1. This call schedules the write to the E²PROM memory and returns immediately. There is no way of knowing when the E²PROM has actually been updated but, barring hardware errors, it will be complete within one second of the call.

E2PromRead

Synopsis

The **E2PromRead** call is made by the application to obtain all or part of the user E²PROM from the card.

C++

```
void E2PromRead (void* UserBuffer,  
                int   BufferLength) ;
```

C#, VB, Java

```
void E2PromRead (Byte[] UserBuffer,  
                int   BufferLength) ;
```

Parameters

1. **UserBuffer**
This is the address of the user's buffer, into which the current contents of the user E²PROM area are copied.
2. **BufferLength**
This is the count of the number bytes to be transferred. If this is greater than 256 the extra will be silently ignored.

Return Value

None

Remarks

1. Unwritten E²Prom memory is initialised all one bits.
2. Writes performed by E2PromWrite will be reflected immediately in the data returned by this function, regardless of whether or not they have been committed to E²Prom memory.

Paylink Event Queue

Introduction

These calls provide access to reports about the detailed workings of the peripherals connected to the system. All Acceptor / Dispenser events such as errors, frauds and rejects (including pass / fail of internal self-test) that are received will be queued (in a short queue) and can be retrieved with API calls in conjunction with an EventDetailBlock object.

Note: Details on the exact make up of the event codes and their classification are given in the “Milan / Paylink System Manual” document, which should be referred to if you are intending to process this data programmatically.

The standard Paylink interface provides a single queue for all events using the **NextEvent()** function.

As this can cause problems in the structure of users applications, the same events can be retrieved on a per acceptor and per dispenser basis using the **NextAcceptorEvent()**, **NextDispenserEvent()** and **NextSystemEvent()** functions.

Note that these two approaches both process the same incoming queue of events from Paylink, and so cannot be mixed - once one of the categorised calls is used, all further events must be retrieved using categorised calls.

There is no intention that these events would be used for the normal operation of the application. Rather, the intention is that they can be captured and presented in “management” reports.

(Obviously, the application can respond automatically to events such as fraud, by disabling everything for a while, but this doesn't have to form part of the algorithms by which the application manages the peripherals.)

The event codes used have an internal structure, allowing categorizations. The bottom 6 bits are the unique event classification code, fault related codes have bit 5 set and otherwise overlap these events code, whilst more significant bits describe the type of unit affected.

For details of the exact makeup of the values of these codes, users are referred to the ImheiEvent.h header file.

Events fall into two categories, notifications and faults. Notifications are just that, the incoming information is passed along to the application.

The fact of a fault on the other hand is remembered by Paylink, and when the fault clears, a NOW_OK “fault” event will be generated.

A specific bit in the event code is reserved for indicating fault events.

The relevant functions calls and structures for these events are below:

NextEvent

Synopsis

This call provides access to reports on the detailed workings of the peripherals connected to the system. All Acceptor / Dispenser events such as errors, frauds and rejects (including pass / fail of internal self-test) that are received will be queued (in a short queue) and can be retrieved with **NextEvent** calls in conjunction with an EventDetailBlock object.

Fields / properties for the EventDetailBlock Object

Field Type	Name	Meaning
int	EventCode	The code (the same as returned by NextEvent)
int	RawEvent	The actual code returned by the peripheral
int	DispenserEvent	Non zero if the device was a dispenser, zero for an acceptor
int	Index	The ReadxxxDetails index of the generating device

```
int NextEvent(EventDetailBlock* EventDetail);
```

Parameters

1. EventDetail
NULL or the address of the single structure at which to store more details of the event given by the return value.

Return Value

The return code is 0 (IMHEI_NULL) if no event is available; otherwise it is the next event.

Remarks

1. In the case where one or more events are missed, the code IMHEI_OVERFLOW will replace the missed events.
2. If only basic information is required, then (as note, coin & Dispenser event codes do not overlap) the **EventDetail** parameter can often be set to NULL, as the device is implicit in the event.
3. The actual values for the **EventCodes** returned are in the separate header file **ImheiEvent.h**
4. The **RawEvent** field for various drivers is often the obvious byte value, but errors (especially self-test errors) on some devices provide more information than will fit into a byte. Details on the exact way in which the raw codes for errors are handled are provided in Appendix 1 and in the event handling section of the system manual.

Driver Software	Raw Code for Event
cctalk coin	Byte from "Read Buffered Credit" response.
cctalk note	Byte from "Read Buffered Bill Events" response.
ID-003	Response to "Status Poll"
CCNet	For Rejected , the reason byte else the response to "Status Poll"
EBDS	The bit number set in the poll reply
MDB Bill Acceptor	Event byte from Poll response
MDB Changer	

NextAcceptorEvent
NextDispenserEvent
NextSystemEvent***Synopsis***

These calls provide controlled access to exactly the same set of events as the **NextEvent** call described above.

The difference is that, rather than providing access to one single queue with all events, these provide access to a number of queues. One independent queue is provided for *each* acceptor in the system, one for each dispenser in the system, and one, final queue for all system oriented events.

```
int NextAcceptorEvent(int Number,  
                     EventDetailBlock* EventDetail);  
int NextDispenserEvent(int Number,  
                      EventDetailBlock* EventDetail);  
  
int NextSystemEvent(EventDetailBlock* EventDetail);
```

Parameters

1. Number
The same value as that used in a call to ReadxxxDetails. All events returned will have an Index value equal to this.
2. EventDetail
NULL, or the address of the single structure at which to store more details of the event given by the return value.

Return Value***Remarks***

1. If these calls are used in a system that also calls **NextEvent**, the result is undefined.
2. Systems with more than 32 acceptors or dispensers should not use these calls.
3. Un-accessed queues will silently discard events.

Utility Functions

CheckOperation (1.11.x)

Synopsis

This call allows an application to check that the Paylink and its connection to the PC are operational. It also allows the application to automatically close down currency acceptance in the event of any PC malfunction.

```
int CheckOperation(int Sequence,  
                  int Timeout);
```

Parameters

1. Sequence
A unique number for this call, freely chosen by the application.
2. Timeout
A time in milliseconds before which another **CheckOperation()** call must be made, *with a different value in **Sequence***, in order to continue the normal operation of Paylink. If zero, then this functionality is inactive from then on.

Return Value

The last **Sequence** value of which the Paylink unit has been notified, or -1 if the Paylink does not support this facility.

Remarks

1. In normal operation, Paylink can be expected to have updated the value to be returned by this within 100 milliseconds of the previous call. It is suggested that this call is made every 500 milliseconds or longer to allow for transient delays.
2. If the **Timeout** expires, Paylink will “silently” disable all the acceptors that are connected to it. The next call to **CheckOperation()** will “silently” re-enable them. This facility is not operation until the first call of **CheckOperation()**.
3. From **1.11.3** onwards, the configuration file can specify an output related to this function. If ther outp is specfied then it is driven *only* when the **Timeout** is being checked, *and* has not expired.
4. Statistics as to how close the system gets to a timeout are output to the Paylink log, which can aid the process of ensuring reliable operation at the application level.

SetDeviceKey

Synopsis

The **SetDeviceKey** call is made by the application to set such things as an encryption key.

Note that this does **not** store the encryption key into the peripheral; it merely notifies Paylink of the key to use for communication.

```
void SetDeviceKey (int InterfaceNo,  
                  int Address,  
                  int Key) ;
```

Parameters

1. InterfaceNo
The Interface on which the device is located
2. Address
The address of the device whose key is being updated
3. Key
The 32 bit key to be remembered for the device.

Return Value

None

Remarks

1. At present, this can only be used for a cctalk BNV acceptor at address 40 on the cctalk interface.
The key (as 6 hex digits) is used as the encryption key.
2. An example application for this is available within the SDK folder structure.

SerialNumber

Synopsis

The **SerialNumber** call provides access to the electronic serial number stored on the Paylink device.

```
int SerialNumber (void) ;
```

Parameters

None

Return Value

32 bit serial number.

Remarks

1. A serial number of -1 indicates that a serial number has not been set in the device.
2. A serial number of 0 indicates that the device firmware does not support serial numbers

FirmwareVersion

Synopsis

The **FirmwareVersion** call allows a control application to discover the exact description of the firmware running on the unit.

C++

```
int FirmwareVersion (char* CompileDate,  
                    char* CompileTime);
```

C#, VB

```
int FirmwareVersion (ref string CompileDate,  
                    ref string CompileTime);
```

Java

```
Int FirmwareVersion (string[] CompileDate,  
                    string[] CompileTime);
```

Parameters

1. CompileDate
This is a 16 byte string that receives a printable version of the date on which the firmware was created.
2. CompileTime
This is a 16 byte string that receives a printable version of the time at which the firmware was created.

Return Values

The firmware version, as a 32 bit integer. This is normally displayed as 4 x 8 bit decimal numbers separated by dots.

Remarks

In 'C', either or both of the character pointers may be null.

USBDriverStatus

Synopsis

The USBDriverStatus call allows an interested application to retrieve the status of the USBDriver program for Paylink system.

```
int DLL USBDriverStatus (void) ;
```

Parameters

None

Return Values

Mnemonic	Value	Meaning
NOT_USB	-1	Interface is to a PCI card
USB_IDLE	0	No driver or other program running
STANDARD_DRIVER	1	The driver program is running normally
FLASH_LOADER	2	The flash re-programming tool is using the link
MANUFACTURING_TEST	3	The manufacturing test tool is using the link
DRIVER_RESTART	4	The standard driver is in the process of exiting / restarting
USB_ERROR	5	The driver has received an error from the low level driver

Remarks

1. For PCI systems this is obviously meaningless and the system returns NOT_USB
2. Be aware that further error statuses may be added. Any response other than STANDARD_DRIVER should be regarded as indicating that the system is not currently functional.

USBDriverExit

Synopsis

The USBDriverExit call allows a control application to request that the USB driver program exits in a controlled manner.

```
void USBDriverExit (void) ;
```

Parameters

None

Return Values

None

Remarks

This sets the USBDriverStatus to DRIVER_RESTART. Driver programs with version 1.0.3.1 or greater will report their exit by changing the USBDriverStatus to USB_IDLE.

For PCI systems this is obviously meaningless and has no effect.

IMHEIConsistencyError

Synopsis

The **IMHEIConsistencyError** call allows an application to check that a transient (hardware) error has not caused corruption of the underlying data structures used to hold the current monetary situation. Although the use of state tables removes the vulnerability of the system to time problems, it increases its vulnerability to *expensive* hardware errors (which could falsely indicate very large money increments.)

C++

```
char* DLL IMHEIConsistencyError(int CoinTime,  
                                int NoteTime) ;
```

C#, VB, Java

```
string DLL IMHEIConsistencyError(int CoinTime,  
                                int NoteTime) ;
```

Parameters

None

1. CoinTime
Default STANDARD_COIN_TIME = 500 msec.
This is the minimum time in milliseconds that will elapse between successive coin insertions. It should be overridden by the application where a fast coin acceptor is in use.
2. NoteTime
Default STANDARD_NOTE_TIME = 5000 msec.
This is the minimum time in milliseconds that will elapse between successive note insertions. It should be overridden by the application where a fast note acceptor is in use.

Return Value

If all the data structures are both consistent and reasonable, the function returns NULL.

If there is any problem an English text message is returned describing the problem.

Remarks

1. A non-NULL return is usually a totally unrecoverable situation.
It is expected that a production application will report the error, and then stop operation.
2. As well as calling this function periodically, it is recommended that it is called after the detection of a credit increase.

Engineering Support

It is not envisaged that application programmers will use these particular functions.

They are included here for completeness, but can be ignored if you are just interfacing application software to a collection of standard peripherals.

WriteInterfaceBlock

Synopsis

The **WriteInterfaceBlock** call sends a “raw” block to the specified interface.

There is no guarantee as to when, in relation to this, regular polling sequences will be sent, except that while the system is *disabled*, the interface card will not put any traffic onto the interface.

C++

```
void WriteInterfaceBlock (int    Interface,
                          void*  Block,
                          int    Length) ;
```

C#, VB, Java

```
void WriteInterfaceBlock (int    Interface,
                          Byte[] Block,
                          int    Length) ;
```

Parameters

1. Interface

The sequence number of the interface that is being accessed.

2. Block

A pointer to program buffer with a raw message for the interface.

This must be a sequence of bytes, with any addresses and embedded lengths required by the peripheral device included. Overheads such as standard checksums will be added by the Paylink.

3. Length

The number of bytes in the message.

Return Value

None

Remarks

Using this function with some interfaces does not make sense, see status returns from **ReadInterfaceBlock**.

ReadInterfaceBlock.

Synopsis

The **ReadInterfaceBlock** call reads the “raw” response to a single **WriteInterfaceBlock**.

C++

```
int ReadInterfaceBlock (int    Interface,
                       void*   Block,
                       int     Length) ;
```

C#, VB, Java

```
int ReadInterfaceBlock (int    Interface,
                       Byte[]  Block,
                       int     Length) ;
```

Parameters

1. Interface
The sequence number of the interface being accessed
2. Block
A pointer to the program buffer into which any response is read.
3. Length
The space available in the program buffer.

Return Values

+ve return values indicate a message has been returned.

Other values are:

-5	INTERFACE_NO_DATA	The handshake has completed, but no data was returned.
-4	INTERFACE_TOO_LONG	Input command is too long
-3	INTERFACE_NON_EXIST	Non command oriented interface (the corresponding WriteInterfaceBlock was ignored)
-2	INTERFACE_OVERFLOW	Command buffer overflow (the corresponding WriteInterfaceBlock was ignored)
-1	INTERFACE_TIMEOUT	Timeout on the interface - no response occurred (The interface will be reset if possible)
0	INTERFACE_BUSY	The response from the WriteInterfaceBlock has not yet been received
> 0		Normal successful response - the number of bytes received and placed into the buffer.

Remarks

1. Repeated calls to **WriteInterfaceBlock** without a successful response are not guaranteed not to overflow internal buffers.
2. The program is expected to “poll” the interface for a response, indicated by a non-zero return value.

DES security (25-12-1)

Background

With the increasing sophistication of fraudulent activity, in 2010 Crane introduced a published scheme for using DES encryption in conjunction with a system of random keys to provide very high security on ccTalk peripherals.

In order to use these high security peripherals to their full, an enhanced Paylink, known as DES Paylink was specified, but never went into production. As a part of its enhancements, this would include a special “DES Button” accessible through a small hole in the case. This DES Button provides important functionality in handling the engineer configuration of a DES system.

Key Exchange

A DES Key is a string of 8 bytes. Details on the DES system are given in Wikipedia and elsewhere.

All DES peripherals have special facility for entering PC exchange mode, so that as a part of the engineer configuration of a DES system the ccTalk peripherals can be asked to generate a new, random, DES key and to return this to the Paylink.

By design, Paylink does not normally query peripherals for their DES key. If the key stored by Paylink for a peripheral is wrong, the application is notified and the green LED flashes at double speed. If a peripheral that has been identified as having a wrong key has generated a new key, then pressing the DES Button on the Paylink will cause Paylink to retrieve this new key and to store it for future use.

Application notification of a peripheral awaiting a valid key is by the bit:
ACCEPTOR_NO_KEY in the status field for acceptors and the value:
PAY_NO_KEY is the status field for hoppers.

Des Lock

A DES system involving Paylink can be basically secured in one of two ways:

1. The PC and Paylink are both in the same secure enclosure.
Here there is no need to provide any security control over the USB connection - access to the USB cable is equivalent to access to the hard disc of the PC, and this level of access cannot be countered by electronic means.
2. The Paylink, or more particularly the USB connection, is accessible from the general cabinet area.
Here a fraud attempt is possible by removing existing USB cable and connecting the Paylink USB socket to the fraudster's laptop.

To prevent this security problem, the PC application can use DES lock. The functions associated with DES lock are described in this section.

(Only) While Paylink is DES locked:

- The PC and Paylink cross check that each other are using the same key.
- The Payout call only works if the key has been matched
- New DES peripherals can not be added without Paylink being first unlocked
- Pressing the DES button deletes all DES keys (peripheral and Paylink) and unlocks Paylink

Some points about the DES Lock system are:

- Updates to existing Paylink applications are optional - although there can be a security risk.
- The DES lock key is provided by the PC, and so can be held on a read only disk system.
- A DES lock aware application will spot if a different Paylink is substituted, **or** if the Paylink is unlocked in order to change the peripherals

DESSetKey

Synopsis

Inform the DLL that of the current key that is to be shared between the PC and Paylink

```
void DESSetKey (char Key[8]) ;
```

Parameters

1. Key
The 8 byte DES key previously applied using the **DESLockSet** function.

Return Value

None.

Remarks

1. The Key should be as unpredictable as possible. Ideally, it will be a random number generated by the application and saved for future use. For system with read only file systems, it could be derived from Processor ID or similar.
2. The **DESStatus** function (see below) will enable the application to determine the success of this function.

DESLockSet

Synopsis

Apply the lock using the key quoted in this function call.

```
void DESLockSet (char Key[8]) ;
```

Parameters

1. Key
The 8 byte DES key chosen by the PC.

Return Value

None

Remarks

If the Paylink is already DES Locked, then this function will not change the key unless **DESSetKey** has already matched the key stored by Paylink.

DESLockClear

Synopsis

Clear any previous applied DES lock.

```
void DESLockSet (void) ;
```

Parameters

None

Return Value

None

Remarks

6. If the Paylink is already DES Locked, then this function will not unlock Paylink unless **DESSetKey** has already matched the key stored by Paylink.
7. This function differs from pressing the DES button in that keys for the existing DES peripherals are not lost. This can therefore be used by application when an engineer wishes to only update a single peripheral.

DESStatus

Synopsis

The DESStatus call provides the current status of the DES lock system.

```
int DESStatus (void) ;
```

Parameters

None

Return Values.

Value	Meaning	Mnemonic
0	The Paylink is unlocked	DES_UNLOCKED
1	DES Key matched by Paylink and PC	DES_MATCH
-1	Not a DES Paylink	DES_NOT
-2	Paylink wrong key	DES_WRONG
-3	DES Key checking is still being performed.	DES_CHECKING
-4	DES Lock is being applied	DES_APPLYING

Remarks

1. Following a call to **DESLockSet**, or **DESSetKey**, the programmer should poll this to check the operation.
2. The Paylink system is only operation when either DES_UNLOCKED or DES_MATCH has been returned by this function.

Appendix 1 - RawEvent Data for errors

For unit *errors*, this appendix details *how* the Paylink system processes the incoming self-test message data to generate the RawEvent field in the EventDetailBlock. For details on “normal” fault events, see the section on events in the system manual.

These are generally reported as IMHEI_NOTE_UNIT_REPORTED_FAULT or IMHEI_COIN_UNIT_REPORTED_FAULT events.

Simple RawEvent data error code handling.

Driver Software	Raw Code for Fault
cctalk coin	1 st byte of “Perform self-test” response.
cctalk note	1 st byte of “Perform self-test” response.
ID-003	The byte following a FAILURE response
CCNet	The Code1 byte following a 47H response
EBDS	If the acceptor handles self-test, this is the index of the first of the 20 integers that has reported an error

MDB RawEvent Data Error Codes

MDB Fault / Self-Test Codes - These come from a changer device as two bytes and are packed into a single byte by the Milan code. The following interprets this byte:

Reported Byte Range	MDB Error Type	MDB Second byte
0x00->0x6F	0x11 - Discriminator	As Reported Byte
0x70->0x7F	0x10 - General Changer	Reported Byte - 0x70
0x80->0xCF	0x12 - Accept Gate	Reported Byte - 0x80
0xE0->0xEE	0x13 - Separator	(Reported Byte - 0xE0) * 2
0xEF	0x14	N/A
0xF0->0xFF	0x15	Reported Byte - 0xF0

BNR RawEvent Error Codes.

The raw error codes reported by Paylink for the BNR are always in the range 0-99 and represent the last two decimal digits of the error value returned by the interface DLL. These values are in XfsConsts.h

The Paylink system knows the returned values that correspond to a working unit, all other values are reported as errors. Specifically:

Note: It is possible that there are errors that are not correctly reported - I've not gone through a complete list to see. Below is what is *supposed* to happen.

Acceptor

For the Acceptor part of the BNR, an error is reported if:

1. A bnr_GetStatus() returns a device Status with a value other than XFS_S_CDR_DS_ON_LINE and XFS_S_CDR_DS_USER_ERROR (which is reported as STACKER_PROBLEM).

2. A `bnr_CashInStart()` causes an `OperationComplete` call back with a value other than `XFS_RC_SUCCESSFUL`.

Dispenser

For a Recycler unit (dispenser) an error is reported if the physical cash unit status is returned as something other than:

XFS_C_CDR_LCU_OK
 XFS_C_CDR_LCU_FULL
 XFS_C_CDR_LCU_HIGH
 XFS_C_CDR_LCU_LOW
 XFS_C_CDR_LCU_EMPTY

BCR / CR RawEvent Error Codes.

The BCR / CR provides a large amount of error information, the table below is derived from

- CR100 TSP182 Issue 0.9.1 and
- Bulk Coin Recycler TSP151 Issue 3.8

and provides details on how Paylink reports the different faults.

All error codes are **always** be reported as:

EventCode = `IMHEI_COIN_UNIT_REPORTED_FAULT`

Raw Code = Value from Code column

Index = BCR Acceptor

In addition:

Blue error codes will be reported with a subsequent event as:

EventCode = `IMHEI_COIN_DISPENSER_REPORTED_FAULT`

Raw Code = Optional Extra Info for Code 26

Optional Extra Info + 100 for Code 27

Index = Corresponding BCR Hopper

Yellow error codes will be reported with a subsequent event as:

EventCode = `IMHEI_COIN_INTERNAL_PROBLEM`

Raw Code = Value from Last Column + Optional Extra Info

Code	Fault	Optional Extra Info		SYS Err	Stat 1	Stat 2	Yellow Offset
0	OK (no fault detected)	0		-	-	0	
1	EEPROM checksum corrupted	1 = Coin acceptor checksum error		CA	100	103	
		2 = Controller checksum error	CR	CH	255	255	
2	Fault on inductive coils	1 to 5 = Validation coil		CA	100	103	
		6 = Singulator belt sensor (sensor missing)		CH	255	255	
		7 = Coin return sensor (sensor missing)		CH	255	255	
		8 = Singulator belt sensor (active but no belt move)		CH	255	255	
		9 = Coin return sensor (active but no belt move)		CH	255	255	
		10 = Coin acceptor wake-up	CR	CA	100	103	
3	Fault on credit sensor	0		CA	100	103	
8	Fault on sorter exit sensors	1 = 4-way sorter (blocked)	BCR	CA	100	101	

		2 = 8-way diverter (blocked)	BCR	DV	100	101	
		3 = 8-way diverter (timeout, coin not seen)	BCR	DV	100	104	
		11 to 18 = Carousel gate	CR	CH	255	255	
9	NVRAM checksum corrupted	0		CH	255	255	
19	Fault on coin return mechanism	0		CA	100	101	
22	Fault on thermistor	0		CA	100	103	
23	Payout motor fault	1 = Singulator (jammed)		MO	255	255	0
		2 = Escalator (jammed)	BCR	MO	255	255	0
		2 = Conveyor (jammed)	CR	MO	255	255	0
		3 = Motorised reject		MO	255	255	0
		4 = Singulator (no tach)		MO	255	255	0
		5 = Escalator (no tach)	BCR	MO	255	255	0
		5 = Conveyor (no tachs)	CR	MO	255	255	0
		6 = Sing. (sensor block)		CH	255	10	0
		7 = Esc. (sensor block)		CH	255	11	0
		8 = Carousel (no tachs)	CR	MO	255	255	0
		9 = Carousel (jammed)	CR	MO	255	255	0
		10 = Coin rotor – fault	CR	CA	100	103	0
		11 = Coin rotor – tachs	CR	CA	100	103	0
		12 = Coin rotor – diameter opto missing	CR	CA	100	103	0
		13 = Coin rotor – park opto missing	CR	CA	100	103	0
		14 = Coin rotor – coin jam	CR	CA	100	101	0
		15 = Coin rotor – excessive jam-jogs	CR	CA	100	101	0
		16 = Singulator belt broken	BCR	MO	255	255	0
		17 = Escalator belt broken	BCR	MO	255	255	0
26	Payout sensor fault	Hopper = 1 to 8 Opto blocked payout		HO	1 to 8	2	
		Hopper = 11 to 18 Opto blocked idle		HO	1 to 8	3	
		Hopper = 21 to 28 Opto short-circuit idle		HO	1 to 8	3	
		Hopper = 31 to 38 Opto short-circuit payout		HO	1 to 8	3	
		Hopper = 41 to 48 Max. current exceeded		HO	1 to 8	4	
		Hopper = 51 to 58 Bad EEPROM checksum		HO	1 to 8	4	
		Hopper = 61 to 68 Power fail on write		HO	1 to 8	4	
27	Level sensor error	1 = Hopper 1		HO	1	4	
		2 = Hopper 2		HO	2	4	
		3 = Hopper 3		HO	3	4	
		4 = Hopper 4		HO	4	4	
		5 = Hopper 5		HO	5	4	
		6 = Hopper 6		HO	6	4	
		7 = Hopper 7	CR	HO	7	4	
		8 = Hopper 8	CR	HO	8	4	
32	Internal comms bad	0 (parallel interface to validator)	CR	CH	255	255	
33	Supply voltage outside operating limits	1 = +24V rail		CH	255	255	
		2 = +5V rail		CH	255	255	
35	D.C.E. fault	0, 1 = Blocked			?	100	
		2 = Broken		CA	100	103	
40	RAM test fail	0		CH	255	255	

48	Slave device not responding	1 = Cashbox missing		CB	250	252	20
		2 = Hopper tray missing		CH	255	255	20
		3 = No hoppers fitted		CH	255	255	20
		4 = Hopper ID mis-match		CH	255	255	20
		5 = Coin acceptor missing		CA	103	1	20
		6 = No hop. lowest coin		HO	255	255	20
		7 = Mixed currency CA		CA	255	255	20
		8 = Mixed currency HO		HO	255	255	20
		9 = I2C communication error	CR	CH	255	255	20
49	Fault on opto sensor	1 = Reserved		-	-	-	40
		2 = Cashbox full sensor		CB	250	251	40
		3 = Wake-up sensor (Avalanche sensor)		CH	255	255	40
		4 = Exit cup full sensor		CH	255	255	40
		5 = Accept flap	CR	CH	255	255	40
		6 = Cashbox flap	CR	CH	255	255	40
		7 = Carousel track opto	CR	CH	255	255	40
		8 = Carousel pusher opto (credit sensor)	CR	CH	255	255	40
		9 = Coin acceptor diameter opto	CR	CA	100	103	40
		10 = Coin acceptor park opto	CR	CA	100	103	40
50	Battery fault	0		CH	255	255	
51	Door open	1 = Singulator door		CH	255	255	60
		2 = Escalator door	BCR	CH	255	255	60
		3 = Carousel lid	CR	CH	255	255	60
52	Microswitch fault	1 = Reject home		CH	255	255	
53	RTC fault	1 = Read / write fail		CH	255	255	
54	Firmware error	1 = CA bad firmware ID		CA	255	255	
		2 = CA firmware too old		CA	255	255	
		3 = HO bad firmware ID		HO	255	255	
		4 = HO firmware too old		HO	255	255	
		5 = HO bad build ID		HO	255	255	
		6 = CA bad build ID		CA	255	255	
		7 = Unhandled CA event		CA	255	255	
		8 = CA programming err.		CA	255	255	
		9 = Bad DB hop. config.		HO	255	3 to 8	
		10 = Prog. checksum err.		CH	255	9	
		11 = Error during upgrade		CH	255	9	
		12 = Stack overflow		CH	255	9	
55	Initialisation error	1 = Flush timeout		CH	255	255	
56	Supply current outside operating limits	1 = +5V rail	CR	CH	255	255	
		2 = Hopper rail	CR	CH	255	255	
		3 = Solenoid rail	CR	CH	255	255	

Disclaimer

This manual is intended only to assist the reader in the use of this product and therefore Aardvark Embedded Solutions shall not be liable for any loss or damage whatsoever arising from the use of any information or particulars in, or any incorrect use of the product. Aardvark Embedded Solutions reserve the right to change product specifications on any item without prior notice