



PRODUCT INFORMATION - INF016-02

APPLICATION DOWNLOAD OPTION FOR ATA AND FTA KESTRELS

INTRODUCTION

This document describes the facilities of and details relating to the API present within Feedback's Kestrel terminal. The API provides a terminal application with a hardware independent collection of software functions - Whose list of facilities range from simple display and keyboard control through to the acquisition and decoding of media scans.

PREREQUISITES

The Kestrel firmware has been designed to allow an MS-DOS style .EXE file to be downloaded and executed; however, it is important to note that the Kestrel does not contain MS-DOS nor any complete emulation of it. In theory, nearly any compiler that can create an 80x86 .EXE file which does not rely on the presence of specific PC architecture, a PC BIOS or MS-DOS may be used. In practice, a small number of compilers have been successfully used; e.g., Borland Turbo C++ , Borland Turbo Pascal and Microsoft C/C++. It is hoped that the list of compatible compilers will increase.

API OVERVIEW

The functions available via the API all reside within the terminal's firmware and can better be described as a Hardware Abstraction Layer (or HAL for short). Conceptually, the HAL may be likened to the BIOS of a PC - The BIOS provides an application access to a number of hardware resources without the application having to know precisely what the hardware configuration is.

In the same way that application programs rely on fixed entry points to the BIOS of a PC, the API provides a fixed method of entry to the HAL functions - Once a method of entry to a particular function has been selected in a given release of firmware, it will continue to use that method in all subsequent firmware releases. This implies that an application written to run under a particular release of firmware should also run, without any rework, on any future release.

One of the design goals of the HAL was to provide sufficient emulation of MS-DOS to allow the startup code, provided by the compiler vendor, to be used without any (or at worst an absolute minimum of) modification. To this aim, very limited support of 'INT 21h' calls is provided - In general, just enough to allow the startup code and crucial library functions to operate. In it's current release, the HAL supports limited emulation of interrupt vector manipulation and memory allocation routines. Future releases of firmware may contain extended emulation so that other compilers can be used.

Access to the HAL is made via direct far calls to specific addresses - All of which are specified in small assembler files that are supplied in both source and object file formats. Header files are also supplied that can be included into the assembler and 'C/C++' source code of an application, allowing the HAL functions to be invoked in the same way as normal library functions.

APPLICATION DOWNLOAD

An application can be downloaded to a terminal, using an extension of the FAMS protocol, via either the RS485 or RS232 link. The download facility is implemented as part of the HAL and can be triggered by any application that wishes to use the facility - Currently, this facility is supported by all standard built in applications.

A user application, once it has been downloaded to a Kestrel, takes preference over any built in applications that the terminal might have. In other words, no facilities provided by the built in applications are available to the downloaded application and any data maintained by them is lost. The replacement application has immediate, and total, control of all resources available; including all memory and any hardware interfaces.

The user application would normally exist as a .EXE file which, by definition, is a relocatable binary image of a program. On the Kestrel, as on an MS-DOS environment, the .EXE file must be copied to its execution area and relocated prior to execution - This relocation process occurs each time the program is run; which, in turn, implies that there must be two copies of the program within the terminal at all times.

RELEASE SPECIFICS

Within the remainder of this document there are details that relate to various releases of the API. The first formal release had an identity of "K0-API-020" and, unless otherwise specified, all information is pertinent to this (and subsequent) releases. Any information that is not relevant to the original release is marked as follows :

#1 - Indicates that the information relates to a facility introduced in "K0-API-030".

This list of 'additions' will grow as facilities are introduced. In general, it can be assumed that a facility provided by one release is available, and relevant, on all subsequent releases.

HAL MODULE AREAS

The HAL contains a collection of self contained, autonomous, modules where each one provides, and maintains, a particular facility. The modules currently contained include :

- Display Maintenance
- Keyboard Scanning
- Sound Production
- Real Time Clock
- Elapsed Time Control
- Periodic Timer Control
- Relay Control
- Visual Indicators (LED) Control
- Opto Input State
- Board Link State
- Media Acquisition (Barcode, Magstripe, Wiegand)
- Media Decoding
- Pulse Acquisition (#1)
- Time Receiver Decoding (#1)
- Generic Host Communication (RS232 and RS485)
- FACT Communication (RS232 and RS485)
- FAMS Communication (RS232 and RS485)
- Generic Auxiliary Communication (RS232 and RS485)
- Configuration Parameters
- Processor Control
- Program Download Control
- Standard Memory Manipulation
- Flat Memory Manipulation
- Firmware Version Control
- System Error Reporting
- General Functions
- CRC Generation

API CONVENTIONS

In general, all requests made by an application to the HAL are made by issuing a far call to a routine using the PASCAL (or FORTRAN) calling convention; i.e., arguments are pushed onto the stack from left to right and are subsequently removed from the stack by the called function. An equivalent Microsoft 'C' function prototype is :

```
extern Returntype __far __pascal ApiFunction([Arg[...]]);
```

All API functions, when requested by an application, guarantee preservation of the registers 'SI, DI, BP, SP, DS, SS and CS'. Additionally, the direction flag may be modified by an API function and, in situations where it is, it will always be left in the CLEAR state; i.e., the instruction 'CLD' will have been executed. All other flags may be corrupt.

All return values supplied by an API function are returned in one of the following register sets :

- AL - Byte values
- AX - Word values
- DX:AX - Long values
- DX:AX - Far pointers

In situations where the HAL generates a signal for an application (i.e., where it calls a 'call-back' function), it will do so by making a far call to the application specified address without passing any parameters. An equivalent Microsoft 'C' function prototype is :

```
extern Returntype __far __pascal AppFunction(void);
```

If any extra information is required by the application, on receipt of a signal, then it must make a subsequent request of the HAL to retrieve it.

The application function, when called via a HAL signal, must guarantee preservation of the registers 'SI, DI, BP, SP, DS, SS and CS'. All other registers and flags may be corrupt.

If a call-back function is required to return a value then it must be returned in one of the following register sets :

- AL - Byte values
- AX - Word values
- DX:AX - Long values
- DX:AX - Far pointers

Certain API functions deal with binary states (e.g., TRUE and FALSE). When one of these states is passed from an application then the API considers it to be FALSE if it's value is zero and TRUE if it has any other value. When an API function returns a binary state then the value of FALSE is zero and the value of TRUE is one.

MODULE : DISPLAY MAINTENANCE

The Display Maintenance module provides support for a display that might be fitted to the terminal. Functions are available to allow an application to determine whether a display is available and, if so, what type it is.

The module only updates the display when explicitly requested to do so, any changes requested prior to the request are buffered up until a 'commit' command is given. This approach, though slightly more troublesome for an application, can help reduce the perceived display flicker that may occur if frequent single character changes are made.

Function : Fitted
 Syntax : int DISPLAY_Fitted(void);
 Arguments : none
 Return Value : TRUE if display is (thought to be) fitted and functional
 Remarks : Checks for the existence of a functional display.

Function : Type
 Syntax : int DISPLAY_Type(void);
 Arguments : none
 Return Value : Value indicating type of display fitted
 Remarks : Returns 0 for none, 1 for LCD.

Function : Reset
 Syntax : void DISPLAY_Reset(void);
 Arguments : none
 Return Value : none
 Remarks : Forces the display to be reset at the next opportunity.

Function : Variant
 Syntax : void DISPLAY_Variant(char Country);
 Arguments : Country specifier, one of :
 0 – British
 1 – USA
 2 – Spanish
 3 – Portugese
 4 – Danish
 5 – Swedish

Return Value : none
 Remarks : Specifies which country specific character set should be used.
 The 'British' variant is assumed by default.

Function : Setup
 Syntax : void DISPLAY_Setup(int Enable);
 Arguments : TRUE if the 'setup' font is to be used
 Return Value : none
 Remarks : Specifies whether the 'setup' font should be used.

Function : Backlight
 Syntax : int DISPLAY_Backlight(int State);
 Arguments : TRUE if backlight is to be enabled
 Return Value : TRUE if backlight was enabled prior to call
 Remarks :

Function : Commit
 Syntax : void DISPLAY_Commit(void);
 Arguments : none
 Return Value : none
 Remarks : Carries out any pending display related updates.

Function : Clear
 Syntax : void DISPLAY_Clear(void);
 Arguments : none
 Return Value : none
 Remarks : Clears all display.
 Cursor co-ords and flash attributes are reset.

Function : Clear_Line
 Syntax : void DISPLAY_Clear_Line(int Line);
 Arguments : Identity of selected line
 Return Value : none
 Remarks : Clears specified line
 Cursor co-ords are positioned at line start and flash attribute is reset.

Function : Clear_Line_1
 Syntax : void DISPLAY_Clear_Line_1(void);
 Arguments : none
 Return Value : none
 Remarks : Clears line one.
 Cursor co-ords are positioned at line start and flash attribute is reset.

Function : Clear_Line_2
 Syntax : void DISPLAY_Clear_Line_2(void);
 Arguments : none
 Return Value : none
 Remarks : Clears line two.
 Cursor co-ords are positioned at line start and flash attribute is reset.

Function : Clear_Eol
 Syntax : void DISPLAY_Clear_Eol(void);
 Arguments : none
 Return Value : none
 Remarks : Clears from the current cursor position to the line end.

Function : Char
 Syntax : void DISPLAY_Char(char Character);
 Arguments : Character to be shown
 Return Value : none
 Remarks : Shows the specified character.
 Character flashes if the MSB is set.

Function : String
 Syntax : void DISPLAY_String(char __far *String);
 Arguments : Pointer to NULL terminated string
 Return Value : none
 Remarks : Shows each character within 'String' up to (but not including) the NULL terminator.
 A character flashes if it's MSB is set.

Function : Block
 Syntax : void DISPLAY_Block(char __far *Block, int MaxLen);
 Arguments : Block - Pointer to block of characters
 MaxLen - Maximum size of block
 Return Value : none
 Remarks : Shows each character within 'Block' up to (but not including) the NULL terminator or for 'MaxLen'
 bytes. A character flashes if it's MSB is set.

Function : Move
 Syntax : void DISPLAY_Move(int X_Pos, int Y_Pos);
 Arguments : X_Pos - Desired X co-ordinate (1..n)
 Y_Pos - Desired Y co-ordinate (1..n)
 Return Value : none
 Remarks : Positions the write cursor at the specified location.

Function : Blink_On
 Syntax : void DISPLAY_Blink_On(void);
 Arguments : none
 Return Value : none
 Remarks : Enables flashing on all subsequently written characters.

Function : Blink_Off
 Syntax : void DISPLAY_Blink_Off(void);
 Arguments : none
 Return Value : none
 Remarks : Disable flashing on all subsequently written characters.

Function : Fill
 Syntax : void DISPLAY_Fill(char Character);
 Arguments : Display 'pad' character
 Return Value : none
 Remarks : Fills all display with the specified character.
 Cursor co-ords and flash attributes remain untouched.

Function : Spaces
 Syntax : void DISPLAY_Spaces(int Amount);
 Arguments : Number of 'space' characters to write
 Return Value : none
 Remarks : Writes the specified number of spaces.

Function : Backspace
Syntax : void DISPLAY_Backspace(void);
Arguments : none
Return Value : none
Remarks : Decrements the cursor co-ord and overwrites the character there with a space.

Function : LimitX
Syntax : int DISPLAY_LimitX(void);
Arguments : none
Return Value : Number of columns supported by display
Remarks : Return value is 'undefined' if no display is fitted.

Function : LimitY
Syntax : int DISPLAY_LimitY(void);
Arguments : none
Return Value : Number of rows supported by display
Remarks : Return value is 'undefined' if no display is fitted.

Function : PosX
Syntax : int DISPLAY_PosX(void);
Arguments : none
Return Value : Current X co-ordinate
Remarks : Return value is 'undefined' if no display is fitted.

Function : PosY
Syntax : int DISPLAY_PosY(void);
Arguments : none
Return Value : Current Y co-ordinate
Remarks : Return value is 'undefined' if no display is fitted.

MODULE : KEYBOARD SCANNING

The Keyboard Scanning module maintains all keyboard scanning related functions. To ensure that the HAL will allow for all current, as well as future, keyboard layouts; only key scancodes are reported. It is the responsibility of the calling application to translate a given scancode into its appropriate key value. The software interface to this module allows for a maximum of 64 different keys.

The module is able to distinguish individual and dual keypress combinations (where a dual keypress combination may be used to provide a shift facility), and also provides an 'auto-repeat' facility.

In general, the module carries out keyboard scanning as a separate background process which, once a keypress has been detected, is suspended until the detected keypress information is removed by the calling application.

Function : Get
 Syntax : int KEYSKAN_Get(void);
 Arguments : none
 Return Value : Composite keyboard scan information
 Remarks : Returns a 16 bit value that can be used to provide a value that can be quickly translated into a key value.

The return value is constructed of :

Return.15	Set if repeated key
Return.14	Clear if >= 1 keypress detected
Return.13	S
Return.12	S Scancode of second keypress
Return.11	S (00..63)
Return.10	S
Return.9	S
Return.8	S
Return.7	Set if two keys pressed
Return.6	Clear if >= 1 keypress detected
Return.5	F
Return.4	F Scancode of first keypress
Return.3	F (00..63)
Return.2	F
Return.1	F
Return.0	F

If no keypress has been detected then the value returned is 0xFFFF; i.e., all bits are set.

Function : Read
 Syntax : int KEYSKAN_Read(char __far *Result)
 Arguments : Pointer to three character array
 Return Value : TRUE if keypress detected
 Remarks : If, and only if, TRUE is returned then the supplied buffer is filled with :

[0] - Repeat count (0x00==First detection)
 [1] - Scancode of first key (MSB set if [2] filled)
 [2] - Scancode of second key (undefined if ([1]&0x80)==0x00)

Note that the repeat count wraps from 0xFF to 0x01.

MODULE : SOUND PRODUCTION

The Sound Production module provides an audible warning facility which allows an application to trigger simple sound sequences.

In general, sounds are produced using a basic 250mS based cycle; where each 'bleep' lasts 250mS and a continuous sound has a 250mS gap between bleeps. This action, when triggered, is maintained by a separate background process.

Within the module, up to 256 volume levels are supported; however, since a user can normally only discern eight different levels, facilities exist to convert from one scale to the other.

Function : Volume
 Syntax : int SOUND_Volume(int Level);
 Arguments : Desired volume level (or 0xFFFF if enquiring current value)
 Return Value : Previous volume level
 Remarks : Sets the volume level (from 0 to 255) for all subsequent sound requests.

Function : Graduate
 Syntax : int SOUND_Graduate(int Level);
 Arguments : Desired volume level (in the range 0 to 7)
 Return Value : Actual volume level setting (in the range 0 to 255)
 Remarks :

Function : Bleep
 Syntax : void SOUND_Bleep(void);
 Arguments : none
 Return Value : none
 Remarks : Triggers a single 'bleep' (of 250mS).

Function : Quantity
 Syntax : void SOUND_Quantity(int Amount)
 Arguments : Number of 'bleeps' required
 Return Value : none
 Remarks : Triggers a series of 'bleeps' (of 250mS on, 250mS off).

Function : Infinity
 Syntax : void SOUND_Infinity(void);
 Arguments : none
 Return Value : none
 Remarks : Triggers a continuous series of 'bleeps' (of 250mS on, 250mS off).

Function : Silence
 Syntax : void SOUND_Silence(void);
 Arguments : none
 Return Value : none
 Remarks : Terminates any current 'bleeping' sequence.

Function : Busy
 Syntax : int SOUND_Busy(void);
 Arguments : none
 Return Value : TRUE if 'bleep' in progress
 Remarks : Return value is 'undefined' if background maintenance is disabled.

Function : Register_On
 Syntax : void SOUND_Register_On(int (__far __pascal *Func)(void));
 Arguments : Pointer to application 'call-back' function
 Return Value : none
 Remarks : The 'call-back' function is called each time the 'bleep' is about to begin.
 If the called application function returns TRUE then the audible alarm is not enabled.
 The 'call-back' facility can be disabled by supplying a NULL address.

Function : Register_Off
 Syntax : void SOUND_Register_Off(int (__far __pascal *Func)(void));
 Arguments : Pointer to application 'call-back' function
 Return Value : none
 Remarks : The 'call-back' function is called each time the 'bleep' is about to end.
 If the called application function returns TRUE then the audible alarm is not disabled.
 The 'call-back' facility can be disabled by supplying a NULL address.

Function : Auto
 Syntax : void SOUND_Auto(int Enable);
 Arguments : TRUE if background processing is to be enabled
 Return Value : none
 Remarks : If the background processing is disabled, then any call relating to the facility becomes unavailable. By default, background processing is enabled.

Function : Tone
 Syntax : void SOUND_Tone(int Duration, int Level);
 Arguments : Duration - Number of 1mS steps
 Level - Desired volume level (0 to 255)
 Return Value : none
 Remarks : A tone of the specified duration and volume is created. Background processing is suspended for the duration of the tone, and the function does not return until the operation is complete.
 Any running or pending background sound is discarded.

Function : Enable
 Syntax : void SOUND_Enable(int Enable, int Level);
 Arguments : Enable - TRUE if sound generator is to be enabled
 Level - Desired volume level (0=Pre-defined level, 1 to 255)
 Remarks : The request is ignored if background processing is enabled.

MODULE : REAL TIME CLOCK

The Real Time Clock module provides access to a battery backed real time clock. The facilities provided include access to the date and time (with one second resolution), a periodic interrupt and a number of date and time related functions.

Communication with this module is centered around a 'Time Block'. This block consists of an eight character array and has the following layout :

```

Block[0] = (Ho) Hours           00..23
Block[1] = (Mi) Minutes         00..59
Block[2] = (Se) Seconds         00..59
Block[3] = (Dw) Day Of Week     01..07 (Monday=1..Sunday=7)
Block[4] = (Dm) Day Of Month    01..31
Block[5] = (Mn) Month           01..12
Block[6] = (Yr) Year            80..79 (1980..2079)
Block[7] = (Rs) Reserved

```

Note : All values are encoded in BCD. The 'Rs' element is reserved for future use. It should be ignored during a read and set to 0xFF during a write.

Function : Read
 Syntax : void CLOCK_Read(char __far *Time);
 Arguments : Pointer to 'Time Block'
 Return Value : none
 Remarks : Fills the supplied array with the current time.
 If no time is available, then all bytes are filled with 0x00 - The 'Dw' field should be used to determine whether the supplied time is valid.

Function : Write
 Syntax : void CLOCK_Write(char __far *Time);
 Arguments : Pointer to 'Time Block'
 Return Value : none
 Remarks : Sets the clock using the contents of the supplied 'Time Block'.
 It is the responsibility of the host to ensure that all the supplied information is valid - Excluding the 'Day Of Week'; which is internally calculated.

Function : Valid
 Syntax : int CLOCK_Valid(char __far *Time, int ChkBCD);
 Arguments : Time - Pointer to 'Time Block'
 ChkBCD - TRUE if BCD legality check is required
 Return Value : TRUE if supplied information is valid
 Remarks : No check is carried out on element 'Rs'.
 The element 'Dw' is only given a range check (of 0..7) - It's correctness (in relation to the supplied date) is not checked.

Function : Zeller
 Syntax : int CLOCK_Zeller(char Day, char Month, char Year);
 Arguments : Day of month (1..31)
 Month of year (1..12)
 Year (00..99 - Expanded to 1980..2079)
 All values should be BCD and should combine to produce a legal date
 Return Value : The day of week (1=Monday..7=Sunday)
 Remarks : Only intended to work in the range 1/1/1980..28/2/2079.

Function : Julian
Syntax : int CLOCK_Julian(char __far *Time);
Arguments : Pointer to 'Time Block'
Return Value : Day of year (1..366)
Remarks : Takes the supplied date (elements 'Dm', 'Mn' and 'Yr') and determines the day of the year; where Jan 1st produces the result one. Leap years are taken into account. If the element 'Dw' is zero then the date is considered to be unknown - In which case zero is returned.

Function : Discard
Syntax : void CLOCK_Discard(void);
Arguments : none
Return Value : none
Remarks : Causes the module to 'discard' it's current date and time so that any future 'read' requests return 'unknown'. This function only operates correctly from (#1).

Function : Salvage
Syntax : void CLOCK_Salvage(void);
Arguments : none
Return Value : none
Remarks : Causes the module to 'salvage' whatever time is available; which may be unusable.

Function : Register
Syntax : void CLOCK_Register(void (__far __pascal *Func)(void));
Arguments : Pointer to application 'call-back' function
Return Value : none
Remarks : The 'call-back' function is called on each second change.
The 'call-back' facility can be disabled by supplying a NULL address.

Function : Ctrl
Syntax : int CLOCK_Ctrl(int Enable);
Arguments : TRUE if periodic interrupts are required
Return Value : TRUE if periodic interrupts were previously enabled
Remarks : May be used to briefly suspend the module's background action - Which is used to trigger a 'call-back' function.

MODULE : ELAPSED TIME CONTROL

The Elapsed Time Control module provides a collection of routines that allow access to a series of timers, with each timer being maintained via a high priority 5mS system clock.

Two basic types of timer exist: There are five 16 bit downward counts that may be set to any desired value - Counting stops when the value reaches zero. There is also a single 32 bit upward count that is reset on power up and may be reset at any time.

All timers may be used by an application; with the proviso that the first 16 bit downward count (whose identity is 0) may be used during certain foreground tasks within the HAL.

Function : Timer
 Syntax : int TICK_Timer(int Number, int Reload);
 Arguments : Number - Downward count identity (0..4)
 Reload - Value to be loaded into count
 Return Value : Timer count value
 Remarks : If 'Reload' is zero, then no load is carried out; i.e., the call effectively just returns the count status.

Function : Pause
 Syntax : void TICK_Pause(int Number, int Delay);
 Arguments : Number - Downward count identity
 Delay - Number of 1mS steps to wait
 Return Value : none
 Remarks : No return occurs until the specified downward count reaches zero.
 The call should only be made with delays in excess of (about) 15mS.

Function : Count
 Syntax : long TICK_Count(int Reset);
 Arguments : TRUE if system ticker count should be reset
 Return Value : System ticker count
 Remarks : The count is incremented each 5mS and has a theoretical time span of approximately 248 days.

MODULE : PERIODIC TIMER CONTROL

The Periodic Timer Control module allows an application to allocate, use and de-allocate a number of timer control blocks. Each block may be used to provide a 16 bit downward counter that is maintained via a low priority 125mS system clock. The counter can be configured to operate in a number of ways and may, therefore, be of use in a wide variety of situations.

Each timer control block consists of the following (expressed in 'C' form) :

```
typedef void (__far __pascal *HalSigV)(void);

typedef volatile struct          /* Timer control block */
{
    Unsigned    Flags;          /* Control/Status flags */
    Unsigned    Count;          /* Timer count */
    Unsigned    Reload;         /* Auto-reload value for 'Count' */
    HalSigV     Lo_Hi;          /* Signal address for LO to HI transition */
    HalSigV     Hi_Lo;          /* Signal address for HI to LO transition */
} Timer;
```

The 'Count' field is the core element of a timer control block. In general, if this field contains zero then the timer is considered to be idle, and if the field contains any other value then the timer is considered to be running. During normal operation, the count decrements on each system clock tick until it reaches a value of zero. A count value of 0xFFFF is considered to be 'infinite'; in which case no decrement of the count occurs.

The 'Flags' field contains a number of single bit control and status elements; and it is usual for modification of these bits to be carried out using bit-manipulation instructions. Only the eight least significant bits are used, as follows :

- Flags.7 : Block allocation flag - Should not be modified by an application
- Flags.6 : If TRUE then the timer control block is maintained on each system clock tick
- Flags.5 : If TRUE then the module will load the 'Count' with 'Reload' on detecting a change in 'Count' from running (high) to idle (low)
To ensure that this facility operates correctly, it is recommended that the 'Count' is initialised with a value of 2
- Flags.4 : If TRUE then the module will make a call to the function 'Hi_Lo' on detecting a change in 'Count' from running (high) to idle (low)
- Flags.3 : If TRUE then the module will make a call to the function 'Lo_Hi' on detecting a change in 'Count' from idle (low) to running (high)
- Flags.2 : Set when the module detects a change in 'Count' from running (high) to idle (low)
- Flags.1 : Set when the module detects a change in 'Count' from idle (low) to running (high)
- Flags.0 : TRUE if the value of 'Count' was non-zero after the last system clock tick maintenance

Function : Allocate
 Syntax : int __far *TIMER_Allocate(int ErrHlt);
 Arguments : TRUE if system error should be generated on exhausted resource
 Return Value : Pointer to idle timer control block
 Remarks : If no control blocks are available, then the function can either generate a system error or return a NULL pointer. Only the 'Flags' field within the supplied control block are guaranteed to be valid - Flags.0 through to Flags.6 are all clear (i.e., FALSE).

Function : Free
 Syntax : void TIMER_Free(int __far *Block);
 Arguments : Pointer to previously supplied timer control block
 Return Value : none
 Remarks : The action is 'undefined' if the supplied address is illegal.

MODULE : RELAY CONTROL

The Relay Control module maintains a number of relays fitted to a terminal. An application may select automatic or manual control over the relay outputs; i.e., an application can configure the module such that a relay output can be directly controlled by that application, or the application can request that the relay output is pulsed for a specified duration in the background. The default action of the module is to provide automatic control.

Function : Auto
 Syntax : void RELAY_Auto(int Line, int Enable);
 Arguments : Line - Identity of relay to be controlled (1..2)
 Enable - TRUE if 'Automatic Maintenance' required
 Return Value : none
 Remarks :

Function : Enable
 Syntax : void RELAY_Enable(int Line, int Level);
 Arguments : Line - Identity of relay to be controlled (1..2)
 Level - TRUE if relay to be activated
 Return Value : none
 Remarks : The request is ignored if 'Automatic Maintenance' is selected.

Function : Toggle
 Syntax : void RELAY_Toggle(int Line);
 Arguments : Identity of relay to be controlled (1..2)
 Return Value : none
 Remarks : The activation state of the specified relay is toggled.
 The request is ignored if 'Automatic Maintenance' is selected.

Function : Pulse
 Syntax : void RELAY_Pulse(int Line, int Duration);
 Arguments : Line - Identity of relay to be controlled (1..2)
 Duration - Time for which relay should be activated (125mS ticks)
 Return Value : none
 Remarks : The request is ignored if 'Manual Maintenance' is selected.

Function : State
 Syntax : int RELAY_State(int Line);
 Arguments : Identity of relay to be interrogated (1..2)
 Return Value : TRUE if the relay is currently activated
 Remarks :

MODULE : VISUAL INDICATOR (LED) CONTROL

The Visual Indicator Control module maintains a number of LEDs fitted to a terminal. An application may select automatic or manual control over the indicators; i.e., an application can configure the module such that an LED can be directly controlled by that application, or the application can request that the LED is pulsed for a specified duration in the background. The default action of the module is to provide automatic control.

Function : Auto
 Syntax : void LED_Auto(int Line, int Enable);
 Arguments : Line - Identity of indicator to be controlled (1..2)
 Enable - TRUE if 'Automatic Maintenance' required
 Return Value : none
 Remarks :

Function : Enable
 Syntax : void LED_Enable(int Line, int Level);
 Arguments : Line - Identity of indicator to be controlled (1..2)
 Level - TRUE if indicator to be illuminated
 Return Value : none
 Remarks : The request is ignored if 'Automatic Maintenance' is selected.

Function : Toggle
 Syntax : void LED_Toggle(int Line);
 Arguments : Identity of indicator to be controlled (1..2)
 Return Value : none
 Remarks : The activation state of the specified indicator is toggled.
 The request is ignored if 'Automatic Maintenance' is selected.

Function : Pulse
 Syntax : void LED_Pulse(int Line, int Duration, int Flash);
 Arguments : Line - Identity of indicator to be controlled (1..2)
 Duration - Time for which indicator should be activated (125mS ticks)
 Flash - TRUE if indicator should flash (@ 4Hz) while enabled
 Return Value : none
 Remarks : The request is ignored if 'Manual Maintenance' is selected.

Function : State
 Syntax : int LED_State(int Line);
 Arguments : Identity of indicator to be interrogated (1..2)
 Return Value : TRUE if the indicator is currently illuminated
 Remarks :

MODULE : OPTO INPUT STATE

The Opto Input State module provides access to a collection of debounced opto inputs. An application can determine, at any time, what state an input is held at and may optionally request an application 'call-back' on detection of a change of state. The latter option could typically be used to provide a simple pulse count facility.

Function : Level
 Syntax : int OPTO_Level(int Line);
 Arguments : Identity of input to be interrogated (1..2)
 Return Value : TRUE if the input is high
 Remarks :

Function : Level_All
 Syntax : int OPTO_Level_All(void);
 Arguments : none
 Return Value : Each bit TRUE if the corresponding input is high
 Remarks : Result.0 represents input one.

Function : Register
 Syntax : void OPTO_Register(int Line, void (__far __pascal *Func)(void));
 Arguments : Line - Identity of associated input (1..2)
 Func - Pointer to application 'call-back' function
 Return Value : none
 Remarks : The 'call-back' function is called each time the state of the associated input changes.
 The 'call-back' facility can be disabled by supplying a NULL address.

MODULE : BOARD LINK STATE

The Board Link State module provides access to a collection of direct inputs (i.e., non-debounced). An application would typically read a non-debounced input during it's startup processing - But may, alternatively, check the state periodically as part of it's normal action.

By convention, the following link identities have been allocated :

- 1 - Diagnostic
- 2 - Keypad type
- 3 - Application selection

Function : State
 Syntax : int LINK_State(int Choice);
 Arguments : Identity of input to be interrogated
 Return Value : TRUE if the specified link is in position
 Remarks :

MODULE : MEDIA ACQUISITION

The Media Acquisition series of modules provide the ability to accumulate all information pertaining to a media scan so that it can subsequently be passed through a suitable decode algorithm. All information accumulated during this process is stored within the module and can be accessed by requesting the information address from that module.

A typical media scan and decode sequence takes the form :

- 1) Trigger the media input
- 2) Wait until a complete scan has been processed
- 3) Fetch scan information address
- 3) Pass scan information through decode algorithm

A media scan sequence, once started, continues in the background - Thus allowing the calling application to continue with other tasks until the scan is complete.

The Kestrel (currently) has hardware sufficient to support three media inputs and the HAL supports four different types of media per input. The 'base-name' of all supported functions include :

MED1BCR_ : Barcode input 1
MED2BCR_ : Barcode input 2
MED3BCR_ : Barcode input 3

MED1EDG_ : Edge input 1
MED2EDG_ : Edge input 2
MED3EDG_ : Edge input 3

MED1MAG_ : Magstripe input 1
MED2MAG_ : Magstripe input 2
MED3MAG_ : Magstripe input 3

MED1WEG_ : Wiegand input 1
MED2WEG_ : Wiegand input 2
MED3WEG_ : Wiegand input 3

Each of the previous modules contains functionally identical facilities; and, as such, are only described once with a base name of MED?Typ_.

Function : Reset
 Syntax : void MED?Typ_Reset(void);
 Arguments : none
 Return Value : none
 Remarks : The media input is forced into it's idle state.
 Any active input is discarded.

Function : Trigger
 Syntax : void MED?Typ_Trigger(int Level);
 Arguments : Polarity level specifier - See 'Remarks'
 Return Value : none
 Remarks : The interpretation of the 'Level' specifier may vary from one input type to another.
 For MED?BCR_; if 'Level' is FALSE then the media input polarity is considered to be BAR!/SPACE - Otherwise, the polarity is considered to be !BAR/SPACE.
 For MED?EDG_; if 'Level' is FALSE then scan information is based upon falling edges - Otherwise it is based upon rising edges.
 For MED?MAG_; if 'Level' is FALSE then data is latched on the clock's falling edge - Otherwise it is latched on the clock's rising edge.
 For MED?WEG_; if 'Level' is FALSE then data is latched on the clock's falling edge - Otherwise it is latched on the clock's rising edge.

Function : Test
 Syntax : int MED?Typ_Test(void);
 Arguments : none
 Return Value : TRUE if media scan complete
 Remarks :

Function : Addr
 Syntax : int __far *MED?Typ_Addr(void);
 Arguments : none
 Return Value : Pointer to media acquisition buffer
 Remarks : The result is only valid once 'MED?Typ_Test' returns TRUE.

MODULE : MEDIA DECODING

The Media Decoding series of modules provide the ability to decode a previously acquired collection of media scan information.

The Kestrel (currently) supports four different types of media decodes; where each type is held in it's own module - In general, each media acquisition type has a complimentary decode module.

Each decode algorithm must be supplied with the following information :

'ScanData'	An array of word values that contains all information pertaining to the media scan.
'Codes'	Specifies which decodes should be used during the decode attempt. An individual bit should be set to enable a particular decode (where bit 0 represents the first code). A list of the relationship between a particular bit and it's decode type can be given by the module.
'Check'	Specifies whether a checksum test should be carried out over the decoded data as part of the decode process. An individual bit should be set to enable the test for a particular decode type. The interpretation of these bits is identical to that of 'Codes'. Not all decodes have a checksum test, and some have an obligatory one - In these situations, any request relating to a checksum test is ignored.
'MaxLen'	Specifies the maximum number of characters that may be present within the decoded data string. If any decode produces a string of too many characters, then the decode is considered to be faulty.

A decode module, when instigated, returns a pointer to a result buffer that may contain the following :

Result[0]	Decode result (0=OK, 1=Checksum Fault, 2=Length Fault, 3=Misread)
Result[1]	Number of characters within decoded data string
Result[2]	Code type (Represents bit position within 'Codes' - May contain a character in the range '0'..'9' or 'A'..'F').
Result[3]	Scan direction ('F'=Forward, 'R'=Reverse)
Result[4..n]	ASCII string of decoded data (Stripped of any checksum)
Result[n+1]	ASCII string terminator (<Nul>)
Result[n+2..3]	<CheckChar><Nul> (The 'CheckChar' is only included if 'Check' specified and permitted)

A caller should only access all fields within 'Result' after verifying that 'Result[0]' specifies that the decode attempt was successful - No guarantee is made regarding the contents of other fields if the decode attempt failed.

The module can, on request, return a pointer to a 'content' string. This string can be used by a caller to produce a decode list and to determine which bit of 'Codes' relates to which decode. The 'content' string contains a number of elements of the form :

[Number](Description)

The 'Number' is a two digit string in the range (and order) ""00" to "15" where the value of the string directly relates to the bit position within 'Codes'.

The 'Description' can contain from zero to twenty characters and provides an identity of the decode represented by the corresponding bit position within 'Codes'. If no characters are present then it implies that no decode is associated with the bit position.

Each element is separated from it's following one by a comma, while the last one is terminated with a <Nul>.

A typical 'content' string might be :

```
[00]CODE 3 OF 9,           ;Codes.0
[01]EAN 8,                ;Codes.1
[02]EAN 13,               ;Codes.2
[03]CODE 93,              ;Codes.3
[04]CODE 128,             ;Codes.4
[05],INTERLEAVED 2 OF 5  ;Codes.5
[06],STANDARD 2 OF 5     ;Codes.6
[07],MAGSTRIPE - TRACK 2 ;Codes.7
[08],                     ;Codes.8 - Unavailable
[09],                     ;Codes.9 - Unavailable
[10],                     ;Codes.10 - Unavailable
[11],                     ;Codes.11 - Unavailable
[12],                     ;Codes.12 - Unavailable
[13],                     ;Codes.13 - Unavailable
[14],                     ;Codes.14 - Unavailable
[15]<Nul>                 ;Codes.15 - Unavailable
```

The 'base-name' of all supported functions include :

```
DECB_ : Barcode decode
DECE_ : Edge input decode
DECM_ : Magstripe input decode
DECW_ : Wiegand input decode
```

Each of the previous modules contains functionally identical facilities; and, as such, are only described once with a base name of DEC?_.

Function : Types
Syntax : char __far *DEC?_Types(void);
Arguments : none
Return Value : Pointer to 'content' string
Remarks : Returns list of all decode types supported.

Function : Decode
Syntax : char __far *DEC?_Decode(int __far *Scan, int Codes, int Check, int Max);
Arguments : Scan - Pointer to 'ScanData', returned by acquisition module
Codes - Decode enable mask
Check - Decode checksum enable mask
Max - Maximum length of decoded data
Return Value : Pointer to a result buffer
Remarks :

Module : Pulse Acquisition (#1)

The Pulse Acquisition module provides the ability to deduce and accumulate pulse width information. It has primarily been included to provide support for the 'Time Receiver Decoding' module (#1), but may be used to acquire any (relatively) slowly changing digital input.

Pulses are resolved with an effective resolution of approximately 81.92uS and can extend to a value of about 2.679S.

The time of each pulse is stored as a 16 bit value, where the least significant bit is used to specify the edge that triggered the store; i.e., a 'zero bit' indicates a rising edge and a 'one bit' indicates a falling edge - A falling edge can be interpreted as 'width of a high pulse'. With the edge specifier masked out, the resultant value represents the pulse width in 40.96uS increments.

No check for buffer overflow is made during storage; but is, instead, made on time removal. The buffer is initially filled with 0x0000 and subsequent pulse times are guaranteed to lie in the range 0x0002 to 0xFFEF. A value of 0xFFF0 is used to represent 'Buffer Overflow' and a value of 0xFFFE or 0xFFFF is used to represent 'Timer Overflow' (i.e., a pulse in excess of 2.679S).

A pulse duration may be approximated from the accumulated time using the formula :

$$(((Time \& 0xFFFE) * 26843) / 65536 * 100) \mu S$$

Note : Due to hardware constraints, the pulse acquisition module shares the same physical resources as used by the third media input. An application must ensure that only one module is enabled for use at a given point in time - This action may change if the HAL is transferred to another hardware platform.

Function: Config (#1)
 Syntax: void PULSE_Config(void);
 Arguments: none
 Return Value: none
 Remarks: The pulse detection logic is initialised and enabled.

Function: Disable (#1)
 Syntax: void PULSE_Disable(void);
 Arguments: none
 Return Value: none
 Remarks :The pulse detection logic is forced into it's idle state.
 Any pending times are discarded.

Function: Fetch (#1)
 Syntax: int PULSE_Fetch(void);
 Arguments: none
 Return Value : 0x0000 if no time is available
 0xFFF0 if buffer has overflowed (requires re-trigger)
 0xFFFE or 0xFFFF if timer overflowed
 Else pulse duration, with polarity in the LSB

Remarks:

Module : Time Receiver Decoding (#1)

The Time Receiver Decoding module allows a time receiver to be connected to the terminal; which can be used to provide an extremely accurate time source.

Once enabled, the module continuously attempts to accumulate and interpret information from the time receiver and, on detecting a valid update, can issue a 'call-back'.

A number of diagnostic facilities are available, which can be of use when a receiver is to be installed in 'difficult' environments.

Note : This module makes use of the 'Pulse Acquisition' module. The limitations imposed by that module are therefore applicable when this module is used.

Function: Config (#1)
 Syntax: void RXTIME_Config(int Source, int Level);
 Arguments: Source - Type of receiver (0=None, 1=MSF/British, 2=DCF/German)
 Level - TRUE if the level of input times requires inversion
 Return Value: none
 Remarks:

Function: Register (#1)
 Syntax: void RXTIME_Register(void (__far __pascal *Func)(void));
 Arguments: Pointer to application 'call-back' function
 Return Value: none
 Remarks: The 'call-back' function is called shortly after a valid time has been received.
 The 'call-back' facility can be disabled by supplying a NULL address.

Function: Disable (#1)
 Syntax: void RXTIME_Disable(void);
 Arguments: none
 Return Value: none
 Remarks: All time reception and decoding is immediately halted.

Function: Read (#1)
 Syntax: void RXTIME_Read(char __far *Time);
 Arguments: Pointer to 'Time Block' (Refer to 'Real Time Clock' module)
 Return Value: none
 Remarks: Should only be called from within a Time Received 'call-back'.
 Fills the supplied array with the time just received - Which is known to be valid.

Function: Status (#1)
 Syntax: int RXTIME_Status(void);
 Arguments: none
 Return Value: Number of contiguous valid seconds of information detected (0..59) or 0xFFFF if decode awaiting synchronisation signal
 Remarks: Can be used for diagnostic purposes.

Function: DiagBit (#1)
 Syntax: char __far *RXTIME_DiagBit(void);
 Arguments: none
 Return Value: Address of internal decode acquisition buffer
 Remarks: Can be used for diagnostic purposes.
 The decode acquisition buffer is implemented as a simple FIFO array of 59 bytes, where data is entered at the last byte; i.e., Bits[58].
 Only the last 'n' bytes are valid - Where 'n' specifies the number of contiguous valid seconds received; as returned by 'Status'.

Function: DiagTime (#1)
 Syntax: int __far *RXTIME_DiagTime(void);
 Arguments: none
 Return Value: Address of internal rotating pulse time buffer
 Remarks: Can be used for diagnostic purposes.
 The format of the pulse time buffer is -

[0] Buffer Size (in bytes)
 [2] Input Index
 [4] Output Index
 [6] Buffer Proper

All elements within the buffer are 16 bit words.
 It is assumed that the application is able to recover data from the buffer faster than it can be entered - No buffer overflow checks are carried out by this module.
 Refer to the Pulse Acquisition module for a description of the time interpretation.

Module : Generic Host Communication

The Generic Host Communication modules allow an application to transfer data via the Kestrel's host communication ports; which, currently, support RS232 and both two and four wire RS485. All incoming and outgoing data is buffered by the modules which thus allow simple and versatile control by an application. The modules also allow for transmit and receive 'call-back'; which allow an application to achieve complex background communication functions.

In general, the facilities supported by the RS232 and RS485 modules are very similar. In the following descriptions the base name HSTtyp is used to describe functions that are available within both - Functions that are only available to one are given their full name (i.e., HST232 for RS232 and HST485 for RS485).

The RS232 module (and hardware interface) support RTS and CTS control lines - The action of these lines cannot be disabled.

The RS485 module (and hardware interface) support both two and four wire operation; i.e., it can be configured for both half and full duplex. When configured for half duplex, the module automatically disables reception while transmission is in progress.

Function: Config
 Syntax: void HSTtyp_Config(int Baud, int Format);
 Arguments: Baud - Communication rate (0=600 Baud .. 6=38400 Baud)
 Format - Made up of:

Format.0 - Stop bits	(0=>1, 1=>2)
Format.1 - Length	(0=>7, 1=>8)
Format.2 - Parity	(00=>None, 01=>Zero)
Format.3 - Parity	(10=>Odd, 11=>Even)
Format.4 - Duplex	(0=>Half, 1=>Full)

Return Value: none
 Remarks: On exit, the channel is still idle.

Function: Disable
 Syntax: void HSTtyp_Disable(void);
 Arguments: none
 Return Value: none
 Remarks: The channel is totally disabled.

Function: Ctrl
 Syntax: int HSTtyp_Ctrl(int Options);
 Arguments: Series of single options, constructed of :
 Options.0 - Disable Tx
 Options.1 - Enable Tx
 Options.2 - Flush Tx buffer
 Options.3 - Reserved
 Options.4 - Disable Rx
 Options.5 - Enable Rx
 Options.6 - Flush Rx buffer
 Options.7 - Clear Rx error
 Return Value: Number of characters pending transmission
 Remarks:

Function: Register_Tx
 Syntax: void HSTtyp_Register_Tx(void (__far __pascal *Func)(void));
 Arguments: Pointer to application 'call-back' function
 Return Value: none
 Remarks: The 'call-back' function is called immediately prior to the module's attempt to remove a character from it's transmit buffer - It is, therefore, the ideal point to attempt insertion of a character for transmission. The 'call-back' facility can be disabled by supplying a NULL address.

Function: Register_Rx
 Syntax: void HSTtyp_Register_Rx(void (__far __pascal *Func)(void));
 Arguments: Pointer to application 'call-back' function
 Return Value: none
 Remarks: The 'call-back' function is called immediately after the modules attempt to add a character to it's receive buffer - It is, therefore, the ideal point to attempt removal of a character. The 'call-back' facility can be disabled by supplying a NULL address.

Function: Transmit
 Syntax: int HSTtyp_Transmit(char Data);
 Arguments: Character to insert to module's transmit buffer
 Return Value: TRUE if character successfully inserted
 Remarks: If no space is available within the module to store the supplied character, then the request is ignored.

Function: Receive
 Syntax: int HSTtyp_Receive(void);
 Arguments: none
 Return Value: 0xFFFF if no character available
 0x8000+Character if character available and error detected
 Character if character available
 Remarks: The module latches onto any receive error detected and informs the caller at the next possible opportunity via this call - An error can be cleared via the 'Ctrl' function.

Module : FACT Communication

The FACT Communication modules provide an application with the low level functionality required within a FACT based application. Modules are available to drive the Kestrel's host communication channels; which (currently) support both RS232 and two wire RS485. All incoming and outgoing data is buffered by the modules. The modules support 'call-back' facilities suitable for the interface selected.

In general, the facilities supported by the RS232 and RS485 modules are very similar. In the following descriptions the base name FACtyp is used to describe functions that are available within both - Functions that are only available to one are given their full name (i.e., FAC232 for RS232 and FAC485 for RS485).

The RS232 module (and hardware interface) support RTS and CTS control lines - The action of these lines cannot be disabled.

A typical FACT application requires a transmit/receive couplet and normally follows a sequence of the form :

- 1) Configure channel
- 2) Synchronise channel
- 3) Transmit message to host
- 4) Wait for response
- 5) Acknowledge response
- 6) Loop to (3) for next 'couplet'

Function: Config
 Syntax: void FACtyp_Config(int Baud, char Term, int Flags);
 Arguments: Baud - Communication rate (0=600 Baud .. 6=38400 Baud)
 Term - Message termination character
 Flags - Made up of :
 Flags.0 – LRC (0=>No, 1=>Yes)
 Flags.1 - Xon/Xoff (0=>No, 1=>Yes)
 Return Value: none
 Remarks: The module is initialised - Any running couplet is aborted.

Function: Poll
 Syntax: void FAC485_Poll(int Minimum, int Stable);
 Arguments: Minimum - Specifies (in uS) how long the line must be in a break condition before it is considered to be a poll
 Stable - Specifies (in uS) how long the line must be in a mark condition, following a 'Minimum' break, before it is considered to be a poll
 Return Value: none
 Remarks: A value of zero, for either argument, is considered to be a request to use the default value for that setting.
 The module should be in an idle state (i.e., not within a 'couplet') when this call is made.

Function: Register
Syntax: void FAC485_Register(void (__far __pascal *Func)(void));
Arguments: Pointer to application 'call-back' function
Return Value: none
Remarks: The 'call-back' function is called on detection of a FACT poll; i.e., immediately prior to a transmission during a 'couplet'.
 The 'call-back' facility can be disabled by supplying a NULL address.

Function: Register_Off
Syntax: void FAC232_Register_Off(void (__far __pascal *Func)(void));
Arguments: Pointer to application 'call-back' function
Return Value: none
Remarks: The 'call-back' function is called on detection of an Xoff character; i.e., on transmission disable.
 The 'call-back' facility can be disabled by supplying a NULL address.

Function: Register_On
Syntax: void FAC232_Register_On(void (__far __pascal *Func)(void));
Arguments: Pointer to application 'call-back' function
Return Value: none
Remarks: The 'call-back' function is called on detection of an Xon character; i.e., on transmission enable.
 The 'call-back' facility can be disabled by supplying a NULL address.

Function: Disable
Syntax: void FACtyp_Disable(void);
Arguments: none
Return Value: none
Remarks: The channel is totally disabled.

Function: Sync
Syntax: void FACtyp_Sync(void);
Arguments: none
Return Value: none
Remarks: Any 'couplet' is aborted and the module returns to it's idle state.

Function: Transmit
Syntax: void FACtyp_Transmit(char __far *Msg);
Arguments: Pointer to <Nul> terminated message
Return Value: none
Remarks: Triggers a communication couplet - Any running one is terminated.
 The <Nul> terminator is replaced with the termination character given during 'Config'.
 The supplied message is truncated to 512 characters prior to transmission.

Function: Status
Syntax: int FACtyp_Status(void);
Arguments: none
Return Value: Module state (i.e., position within 'couplet') that can be :
0 - Module idle
1 - Waiting for poll (to trigger transmission)
2 - Transmitting message
3 - Switching off transmission
4 - Waiting for first character of response
5 - Receiving remainder of response
6 - Pending acknowledgement of received response

Remarks:

Function: Receive
Syntax: char __far *FACtyp_Receive(void);
Arguments: none
Return Value: Pointer to received message
Remarks: If no response received, then NULL is returned.
The caller should copy the response to a local store as soon as possible after making this call.

Module : FAMS Communication

The FAMS Communication modules provide an application with the low level functionality required within a FAMS based application. Modules are available to drive the Kestrel's host communication channels; which (currently) support both RS232 and both two and four wire RS485. All incoming and outgoing data is buffered by the modules. The modules support 'call-back' facilities suitable for the interface selected.

In general, the facilities supported by the RS232 and RS485 modules are very similar. In the following descriptions the base name FAMtyp is used to describe functions that are available within both - Functions that are only available to one are given their full name (i.e., FAM232 for RS232 and FAM485 for RS485).

The RS232 module (and hardware interface) support RTS and CTS control lines - The action of these lines cannot be disabled.

A typical FAMS application requires a receive/transmit couplet and normally follows a sequence of the form :

- 1) Configure channel
- 2) Wait for reception of message
- 3) Trigger transmission of terminal response
- 4) Loop to (2) for next 'couplet'

All communication maintained by these modules is generally maintained as a separate background process; i.e., once the module is initiated, an application is triggered into action (via a 'call-back') when a valid message header is received. Normally the 'call-back' function would generate and trigger the transmission of a response.

The modules allow for optional data to follow a command header and, to support this facility, they maintain a timer (with 1mS resolution) that can be used for timeout error detection.

While one of the modules transmits a response, reception is disabled, and the receive buffer will be purged prior to reception being re-enabled; i.e., The modules are suitable for both two and four wire RS485 operation.

The modules are optimised for, and directly support, detection of FAMS style commands and are not capable of detecting any other form of message. They can also be configured to detect the 'Single Byte Poll' facility - When one is detected, the module creates a 'normal' full message which is, in turn, passed onto the application - In this situation, the only difference is that the length byte, within the constructed message, specifies one.

Function: Config
 Syntax: void FAMtyp_Config(int Addr, int Baud, int Poll);
 Arguments: Addr - Terminal address (in range 0x00..0xFF)
 Baud - Communication rate (0=600 Baud .. 6=38400 Baud)
 Poll - Made up of:

Poll.0 - 'D' recognition [0xC0..0xFD] (1=>Enable)
 Poll.1 - 'd' recognition [0x9A..0xBF] (1=>Enable)

Return Value: none
 Remarks: The module, though active, will not pass on any received commands until a 'call-back' address has been supplied.
 A terminal address of 0xFE or 0xFF has the effect of limiting acceptable receive commands to 'broadcasts'.

Function: Register
Syntax: void FAMtyp_Register(void (__far __pascal *Func)(void));
Arguments: Pointer to application 'call-back' function
Return Value: none
Remarks: The 'call-back' function is called on detection of a FAMS command.
 The 'call-back' facility can be disabled by supplying a NULL address.

Function: Delay
Syntax: void FAMtyp_Delay(int Delay);
Arguments: Delay between receipt of response (from application) and initiation of transmission (as the number of 1mS steps)
Return Value: none
Remarks: The supplied value specifies the minimum message turnaround time of the terminal.

Function: Disable
Syntax: void FAMtyp_Disable(void);
Arguments: none
Return Value: none
Remarks: The channel is totally disabled.

Function: Ctrl
Syntax: int FAMtyp_Ctrl(int Enable);
Arguments: TRUE if command detection to be enabled
Return Value: Command detection state prior to call
Remarks: Can be used to temporarily suspend background command detection.

Function: RxAddr
Syntax: char __far *FAMtyp_RxAddr(void);
Arguments: none
Return Value: Address of pending received command
Remarks: Returns NULL if no pending command available.
 Any pending command must be acknowledged before another command can be detected.

Function: Receive
Syntax: int FAMtyp_Receive(void);
Arguments: none
Return Value: Character received after header - or 0xFFFF if none available
Remarks: Between reception of a command and transmission of a response, the module stores all received characters in a small local buffer.

Function: Timer
Syntax: int FAMtyp_Timer(int Reset);
Arguments: TRUE if count to be reset
Return Value: Receive timeout count
Remarks: Count is automatically started on detection of a command and becomes unavailable on instigation of a transmission.

Function: Transmit
Syntax: void FAMtyp_Transmit(int Action, int Qty, char __far *Data);
Arguments: Action - Required module action
 Qty - Amount of data to transmit
 Data - Pointer to start of data block or the address of a 'call-back'
Return Value: none
Remarks: Should only be called during a command seen 'call-back'.
 If Action=0 then no transmission occurs and command reception is restarted.
 If Action=1 then the data block is transmitted prior to the restart of command reception - There must be at least one character sent.
 If Action=2 then the 'call-back' is repeatedly called to fetch data for transmission until it returns the value 0xFFFF - There must be at least one character sent.

Function: ForceTx (#1)
Syntax: int FAMtyp_ForceTx(int Action, int Qty, char __far *Data);
Arguments: Action - Required module action
 Qty - Amount of data to transmit
 Data - Pointer to start of data block or the address of a 'call-back'
Return Value: TRUE if request successful
Remarks: Requests an immediate transmission of a data block prior to the restart of normal command reception.
 If Action=1 then the data block is transmitted - There must be at least one character sent.
 If Action=2 then the 'call-back' is repeatedly called to fetch data for transmission until it returns the value 0xFFFF - There must be at least one character sent.
 The request fails if the module is already executing a communication sequence.

Function: SetAddr (#1)
Syntax: void FAMtyp_SetAddr(int Addr);
Arguments: Addr - Terminal address (in range 0x00..0xFF)
Return Value: none
Remarks: Changes the terminal address - A value of 0xFE or 0xFF has the effect of limiting acceptable receive commands to 'broadcasts'.

Module : Generic Auxiliary Communication

The Generic Auxiliary Communication modules allow an application to transfer data via the Kestrel's auxiliary communication ports; which, currently, support RS232 and two wire RS485. All incoming and outgoing data is buffered by the modules which thus allow simple and versatile control by an application. The modules also allow for transmit and receive 'call-back'; which allow an application to achieve complex background communication functions.

In general, the facilities supported by the RS232 and RS485 modules are very similar. In the following descriptions the base name AUXtyp is used to describe functions that are available within both - Functions that are only available to one are given their full name (i.e., AUX232 for RS232 and AUX485 for RS485).

The RS232 module (and hardware interface) support RTS and CTS control lines - The action of these lines cannot be disabled.

The RS485 module (and hardware interface) supports only two wire operation; i.e., it only supports half duplex. The module automatically disables reception while transmission is in progress.

Function: Config
 Syntax: void AUXtyp_Config(int Baud, int Format);
 Arguments: Baud - Communication rate (0=600 Baud .. 6=38400 Baud)
 Format - Made up of :
 Format.0 - Stop bits (0=>1, 1=>2)
 Format.1 - Length (0=>7, 1=>8)
 Format.2 - Parity (00=>None, 01=>Zero)
 Format.3 - Parity (10=>Odd, 11=>Even)
 Return Value: none
 Remarks: On exit, the channel is still idle.

Function: Disable
 Syntax: void AUXtyp_Disable(void);
 Arguments: none
 Return Value: none
 Remarks: The channel is totally disabled.

Function: Ctrl
 Syntax: int AUXtyp_Ctrl(int Options);
 Arguments: Series of single options, constructed of :
 Options.0 - Disable Tx
 Options.1 - Enable Tx
 Options.2 - Flush Tx buffer
 Options.3 - Reserved
 Options.4 - Disable Rx
 Options.5 - Enable Rx
 Options.6 - Flush Rx buffer
 Options.7 - Clear Rx error
 Return Value: Number of characters pending transmission
 Remarks:

Function: Register_Tx
Syntax: void AUXtyp_Register_Tx(void (__far __pascal *Func)(void));
Arguments: Pointer to application 'call-back' function
Return Value: none
Remarks: The 'call-back' function is called immediately prior to the module's attempt to remove a character from its transmit buffer - It is, therefore, the ideal point to attempt insertion of a character for transmission.
 The 'call-back' facility can be disabled by supplying a NULL address.

Function: Register_Rx
Syntax: void AUXtyp_Register_Rx(void (__far __pascal *Func)(void));
Arguments: Pointer to application 'call-back' function
Return Value: none
Remarks: The 'call-back' function is called immediately after the module's attempt to add a character to its receive buffer - It is, therefore, the ideal point to attempt removal of a character.
 The 'call-back' facility can be disabled by supplying a NULL address.

Function: Transmit
Syntax: int AUXtyp_Transmit(char Data);
Arguments: Character to insert to module's transmit buffer
Return Value: TRUE if character successfully inserted
Remarks: If no space is available within the module to store the supplied character, then the request is ignored.

Function: Receive
Syntax: int AUXtyp_Receive(void);
Arguments: none
Return Value: 0xFFFF if no character available
 0x8000+Character if character available and error detected
 Character if character available
Remarks: The module latches onto any receive error detected and informs the caller at the next possible opportunity via this call - An error can be cleared via the 'Ctrl' function.

Module : Configuration Parameters

The Configuration Parameter module provides a relatively small area of non-volatile storage that can be used, by an application, to store application specific configuration data. Facilities are provided by the module to allow an application to make changes to a number of parameters prior to them being committed; i.e., any power failure that occurs during the update of the configuration data can be 'reversed' on a subsequent power-up.

Function:	Read
Syntax:	char CONF_Read(int Address);
Arguments:	Address - Address of configuration parameter (0..511)
Return Value:	Configuration parameter requested
Remarks:	
Function:	Write
Syntax:	void CONF_Write(int Address, char Value);
Arguments:	Address - Address of configuration parameter (0..511) Value - Value of new parameter
Return Value:	none
Remarks:	The written value is not stored until subsequently 'committed'; i.e., A read of the same address would return the old value until a 'commit' has been requested.
Function:	Store
Syntax:	void CONF_Store(int Address, char Value);
Arguments:	Address - Address of configuration parameter (0..511) Value - Value of new parameter
Return Value:	none
Remarks:	The specified parameter, and all previously uncommitted parameters, are committed immediately prior to return.
Function:	Sync
Syntax:	void CONF_Sync(void);
Arguments:	none
Return Value:	none
Remarks:	Any previously uncommitted parameter writes are discarded by this call.
Function:	Commit
Syntax:	void CONF_Commit(void);
Arguments:	none
Return Value:	none
Remarks:	Any previously uncommitted parameter writes are committed.
Function:	Reset
Syntax:	void CONF_Reset(void);
Arguments:	none
Return Value:	none
Remarks:	All configuration parameters are set to 0x00 - And committed.

Module : Processor Control

The Processor Control module provides access to a number of processor and HAL related functions - The majority of which are used internally by the HAL and are made available for system debug purposes. The module also includes a number of 'miscellaneous' facilities which directly relate to processor actions.

Function:	Lock
Syntax:	void PROC_Lock(void);
Arguments:	none
Return Value:	none - Does not return
Remarks:	Causes the processor to 'halt' - In preparation for a power failure. May be used to trigger a forced terminal reset - Assuming that the Kestrel's watchdog link is fitted.
Function:	Watchdog_Trigger
Syntax:	void PROC_Watchdog_Trigger(void);
Arguments:	none
Return Value:	none
Remarks:	Pulses the Kestrel's watchdog timer - This action is normally carried out directly by the HAL.
Function:	Pause
Syntax:	void PROC_Pause(int Quantity);
Arguments:	Duration of pause (as the number of 1uS ticks)
Return Value:	none
Remarks:	Causes a software delay of the specified duration - Should only be considered an approximation.
Function:	Disable_Ints
Syntax:	void PROC_Disable_Ints(void);
Arguments:	none
Return Value:	none
Remarks:	Disables processor interrupts - Use with care.
Function:	Enable_Ints
Syntax:	void PROC_Enable_Ints(void);
Arguments:	none
Return Value:	none
Remarks:	Enables processor interrupts - Use with care.
Function:	Parameters
Syntax:	int PROC_Parameters(int Choice);
Arguments:	Identity of parameter type required - HAL specific, may be : 0 - Segment address of _MCB 1 - Segment address of _FREE 2 - Segment address of VERSION 3 - Segment address of _MAP 4 - Segment address of STACK 5 - Size of STACK
Return Value:	Requested parameter value
Remarks:	

Function: Hold
 Syntax: void PROC_Hold(void);
 Arguments: none
 Return Value: none
 Remarks: All maskable interrupts with a priority of less than 0 (i.e., 1, 2 and 3) are disabled.
 Should be used with extreme brevity.

Function: Release
 Syntax: void PROC_Release(void);
 Arguments: none
 Return Value: none
 Remarks: Reverses action of 'Hold'.
 Should be used with extreme brevity.

Function: Mains
 Syntax: int PROC_Mains(void);
 Arguments: none
 Return Value: TRUE if terminal currently powered by the mains
 Remarks: Only of use on terminals fitted with a UPS.

Module : Program Download Control

The Program Download Control module maintains the download, and subsequent run, of a user application. In brief, an application can trigger the request of a replacement application and it can discard itself (i.e., self destruct) so that one of the terminal's inbuilt applications may be executed.

Function: Trigger
 Syntax: int DOWNLOAD_Trigger(char __far *ConfBlk);
 Arguments: Pointer to download configuration control block of the form :
 ConfBlk[0] - Terminal identity (0x00..0xFD)
 ConfBlk[1] - Interface (0x00=>RS232, 0x01=>RS485)
 ConfBlk[2] - Communication rate (0=>600..6=>38400)
 ConfBlk[3] - Receive timeout multiplier (50mS step)
 ConfBlk[4] - Transmit delay multiplier (1mS step)
 ConfBlk[5] - Transmit terminate (0x00=>No, 0x01=>Yes)
 ConfBlk[6] - 0x00 (Reserved)
 ConfBlk[7] - 0x00 (Reserved)
 ConfBlk[8] - 0x00 (Reserved)
 ConfBlk[9] - Default parameters (0x00=>No, 0x01=>Yes)
 Return Value: TRUE if the trigger request is successful
 Remarks: If 'Defaults' are requested (in 'ConfBlk[9]') then the HAL defaults will be used for the download instead of the parameters supplied.
 If the request succeeds, the calling application should tidy itself up and request a terminal reset.

Function: Discard
 Syntax: void DOWNLOAD_Discard(int Foreground);
 Arguments: TRUE if caller is running in the 'Foreground'
 Return Value: none - Does not return
 Remarks: The previously downloaded application (i.e., the caller) is discarded and all download configuration parameters are reset.
 If the caller is running in the 'Foreground' then a message reporting the action is shown.
 The function does not return but, instead, requests a terminal reset.

Module : Standard Memory Manipulation

The Standard Memory Manipulation module provides basic information pertaining to RAM available within the normal one megabyte address space of an 8086 compatible processor. The routines available allow an application to determine the amount of RAM and carry out tests over it.

When a .EXE file is being executed the HAL provides emulation of certain MS-DOS memory allocation function (via INT 21h). It is recommended that an application requiring dynamic memory allocation should use the DOS compatible facilities.

Function: Size
Syntax: char __far *MEM_Size(void);
Arguments: none
Return Value: Address of last location available within the first one meg address space
Remarks:

Function: Test
Syntax: int MEM_Test(char __far *Block, long Len);
Arguments: Block - Pointer to start of memory block
 Len - Size of memory block (in bytes)
Return Value: TRUE if test successful
Remarks: Carries out a fully destructive test over the specified block.
 On successful completion, the block is filled with <Nul>s.

Module : Flat Memory Manipulation

The Flat Memory Manipulation module provides comprehensive information pertaining to all the memory available to a Kestrel - Which can theoretically encompass up to sixteen megabytes. Facilities are available that allow an application to dynamically allocate memory blocks beyond the first one megabyte.

In general, the memory beyond one megabyte is not directly available to an application; i.e., standard compilers cannot normally access it. An application would normally write to and read from this area using the facilities provided by this module.

All addresses passed to and from this module assume a 'flat' address space and are passed via long integers (i.e., a 32 bit value).

Function: Parameter
 Syntax: long FMEM_Parameter(int Choice);
 Arguments: Identity of parameter type required, may be :
 0 - Reserved
 1 - Lower limit of expansion memory block
 2 - Upper limit of expansion memory block
 3 - Size of expansion memory block

Return Value: Requested parameter value

Remarks:

Function: BlkFill
 Syntax: void FMEM_BlkFill(char 'flat' *Addr, long Qty, char Fill);
 Arguments: Addr - Pointer to start of memory block
 Qty - Size of memory block
 Fill - Pad character for memory block
 Return Value: none
 Remarks: The specified memory block is filled with the specified pad character.

Function: BlkCpyLH
 Syntax: void FMEM_BlkCpyLH(char 'flat' *Dst, char 'flat' *Src, long Qty);
 Arguments: Dst - Pointer to start of destination block
 Src - Pointer to start of source block
 Qty - Size of memory block
 Return Value: none
 Remarks: Copies block from 'Src' to 'Dst' working from the start of the block upwards - Requires consideration if any block overlap exists.

Function: BlkCpyHL
 Syntax: void FMEM_BlkCpyHL(char 'flat' *Dst, char 'flat' *Src, long Qty);
 Arguments: Dst - Pointer to start of destination block
 Src - Pointer to start of source block
 Qty - Size of memory block
 Return Value: none
 Remarks: Copies block from 'Src' to 'Dst' working from the end of the block downwards - Requires consideration if any block overlap exists.

Function: BlkCpy
 Syntax: void FMEM_BlkCpy(char 'flat' *Dst, char 'flat' *Src, long Qty);
 Arguments: Dst - Pointer to start of destination block
 Src - Pointer to start of source block
 Qty - Size of memory block
 Return Value: none
 Remarks: Copies block from 'Src' to 'Dst' - The routine takes account of any block overlap, and acts accordingly.

Function: Test
 Syntax: int FMEM_Test(char 'flat' *Block, long Len);
 Arguments: Block - Pointer to start of memory block
 Len - Size of memory block (in bytes)
 Return Value: TRUE if test successful
 Remarks: Carries out a fully destructive test over the specified block.
 On successful completion, the block is filled with <Nul>s.

Function: Allocate
 Syntax: char 'flat' FMEM_Allocate(long Amount);
 Arguments: Size of requested block
 Return Value: Pointer to allocated block - Or NULL if unsuccessful
 Remarks:

Function: Release
 Syntax: void FMEM_Release(char 'flat' *Block);
 Arguments: Pointer to previously allocated block
 Return Value: none
 Remarks: The result of an attempt to release a non-existent block is undefined - Unless the supplied address is NULL, in which case the request is ignored.

Function: Reallocate
 Syntax: char 'flat' *FMEM_Reallocate(char 'flat' *Block, long Amount);
 Arguments: Block - Pointer to previously allocated block
 Amount - New size of previously allocated block
 Return Value: Pointer to allocated block - Or NULL if unsuccessful
 Remarks: If 'Block' is NULL then an attempt to allocate a new block is made.

Function: Maximum
 Syntax: long FMEM_Maximum(char 'flat' *Block);
 Arguments: Pointer to previously allocated block
 Return Value: Maximum possible size of previously allocated block
 Remarks: If 'Block' is NULL then the maximum size of a newly allocated block is determined.

Module : Firmware Version Control

The Firmware Version Control module allows an application to enquire and test items specific to any firmware fitted to a Kestrel - Which may be of use for diagnostic purposes

Function: Check
 Syntax: int VERSION_Check(void);
 Arguments: none
 Return Value: TRUE if test successful
 Remarks: Carries out a simple checksum test over the main terminal EPROM.

Function: Assy
 Syntax: char __far *VERSION_Assy(void);
 Arguments: none
 Return Value: Pointer to firmware assembly number string
 Remarks: The assembly number string takes the form :
 K0-API-xxx WW.YY

Module : System Error Reporting

The System Error Reporting module provides the terminal, and any application running on it, with a consistent method of reporting fatal errors. When an error is reported, the terminal displays a message (if a display is available) and enters an infinite loop where it beeps for a predefined amount, pauses, and then loops to beep again.

The module also supports a 'call-back' function - This would typically allow an application to tidy itself up immediately prior to the error routines infinite loop.

Function: Register
 Syntax: void ERROR_Register(void (__far __pascal *Func)(void));
 Arguments: Pointer to application 'call-back' function
 Return Value: none
 Remarks: The 'call-back' function is called on entry to an error report.
 The 'call-back' facility can be disabled by supplying a NULL address.

Function: Report
 Syntax: void ERROR_Report(int Fault, char __far *Msg);
 Arguments: Fault - Error number
 Msg - Pointer to (optional) error message
 Return Value: none - Does not return
 Remarks: If 'Msg' is non-NULL then a message of the following form is shown :
 'E nn(Message)"
 Where 'nn' is the specified 'Fault' value and 'Message' normally has the general form ' - (Text)'
 Once the message is shown (assuming a display is available), the terminal beeps for (Fault/10)
 times, pauses for two seconds and loops to re-beep.

Module : General Functions

The General Functions module contains a collection of 'helper' functions that may be of use to applications running on the terminal. A number of the facilities available can also be found, in different forms, within the run time libraries of various compilers.

- Function: Ascii
 Syntax: char __far *GEN_Ascii(int Value);
 Arguments: Integer to be converted to ASCII string
 Return Value: Pointer to ASCII decimal <Nul> terminated string
 Remarks: The string produced is stripped of all leading zeros.
- Function: Itoa
 Syntax: char __far *GEN_Itoa(int Value, char __far *Buf, int Digits);
 Arguments: Value - Integer to be converted to an ASCII decimal string
 Buf - Destination for produced string
 Digits - Number of digits to fill
 Return Value: Pointer to produced string
 Remarks: If 'Buf' is NULL then the routine uses it's own internal transfer buffer.
 No <Nul> terminator is appended to a supplied buffer; but is if the internal transfer buffer is selected.
 If the converted string is longer than 'Digits', then the whole buffer is padded with asterisks.
- Function: ItoH
 Syntax: char __far *GEN_ItoH(int Value, char __far *Buf, int Digits);
 Arguments: Value - Integer to be converted to an ASCII hexadecimal string
 Buf - Destination for produced string
 Digits - Number of digits to fill
 Return Value: Pointer to produced string
 Remarks: If 'Buf' is NULL then the routine uses it's own internal transfer buffer.
 No <Nul> terminator is appended to a supplied buffer; but is if the internal transfer buffer is selected.
- Function: StrPos
 Syntax: int GEN_StrPos(char __far *Buf, char Search);
 Arguments: Buf - String to be searched
 Search - Character to search for
 Return Value: Offset of 'Search' within 'Buf' plus one, or 0 if not found
 Remarks:
- Function: Strcpy
 Syntax: int GEN_Strcpy(char __far *Dst, char __far *Src, int Qty);
 Arguments: Dst - Pointer to start of destination block
 Src - Pointer to start of source block
 Qty - Size of memory block (or 0 if direct string copy)
 Return Value: Number of characters copied (including possible <Nul>)
 Remarks: Copies upwards - Requires consideration if any block overlap exists.
- Function: Strlen
 Syntax: int GEN_Strlen(char __far *String);
 Arguments: Pointer to <Nul> terminated string
 Return Value: Length of string (excluding <Nul>)
 Remarks:

Function: BlkCpyLH
Syntax: void GEN_BlkCpyLH(char __far *Dst, char __far *Src, int Qty);
Arguments: Dst - Pointer to start of destination block
 Src - Pointer to start of source block
 Qty - Size of memory block
Return Value: none
Remarks: Copies block from 'Src' to 'Dst' working from the start of the block upwards - Requires consideration if any block overlap exists.

Function: BlkCpyHL
Syntax: void GEN_BlkCpyHL(char __far *Dst, char __far *Src, int Qty);
Arguments: Dst - Pointer to start of destination block
 Src - Pointer to start of source block
 Qty - Size of memory block
Return Value: none
Remarks: Copies block from 'Src' to 'Dst' working from the end of the block downwards - Requires consideration if any block overlap exists.

Function: PtrNorm
Syntax: void __far *GEN_PtrNorm(void __huge *Addr);
Arguments: Generic pointer
Return Value: Normalised pointer address
Remarks: Result organised such that (Offset&0xFFF0)==0x0000.

Function: PtrDiff
Syntax: long GEN_PtrDiff(void __huge *Adr1, void __huge Adr2);
Arguments: Adr1 - Generic pointer
 Adr2 - Generic pointer
Return Value: Result of 'Adr1'-'Adr2'
Remarks:

Function: PtrFlat
Syntax: long GEN_PtrFlat(void __huge *Addr);
Arguments: Generic pointer
Return Value: An un-segmented 'flat' address of the supplied value
Remarks:

Function: PtrUndo
Syntax: void __far *GEN_PtrUndo(long Addr);
Arguments: Un-segmented 'flat' address
Return Value: Normal Seg:Off version of the supplied value
Remarks: The produced Seg:Off result is not normalised.

Function: HexVal
Syntax: int GEN_HexVal(char Hex);
Arguments: A hexadecimal character ('0'..'9', 'A'..'F')
Return Value: The weight of the character plus one - Or zero if not valid
Remarks:

Function: ChkBCD
Syntax: int GEN_ChkBCD(char Value, char Limit);
Arguments: Value - Character to be checked
Limit - Upper limit of BCD value
Return Value: TRUE if the character is both valid and not greater than 'Limit'
Remarks:

Function: BCDtol
Syntax: int GEN_BCDtol(char __far *String, int Qty);
Arguments: String - Pointer to first of a series of packed BCD characters
Qty - Number of BCD characters in 'String'
Return Value: Integer value of characters within BCD string
Remarks: All supplied BCD characters must be valid on entry.

Function: ItoBCD
Syntax: int GEN_ItoBCD(int Value);
Arguments: Integer value requiring conversion
Return Value: Four packed BCD characters
Remarks:

Function: BlkFill
Syntax: void GEN_BlkFill(char __far *Addr, long Qty, char Fill);
Arguments: Addr - Pointer to start of memory block
Qty - Size of memory block
Fill - Pad character for memory block
Return Value: none
Remarks: The specified memory block is filled with the specified pad character.

Function: FamChkF
Syntax: void GEN_FamChkF(char __far *Src, int Qty, char __far *Dst);
Arguments: Src - Pointer to start of memory block
Qty - Size of memory block
Dst - Pointer to three byte destination
Return Value: none
Remarks: Calculates a 'Full' FAMS checksum over the supplied memory block.
Checksum produced uses modulus 1000000 arithmetic.

Function: FamChkL
Syntax: void GEN_FamChkL(char __far *Src, int Qty, char __far *Dst);
Arguments: Src - Pointer to start of memory block
Qty - Size of memory block
Dst - Pointer to three byte destination
Return Value: none
Remarks: Calculates a 'Limited' FAMS checksum over the supplied memory block.
Checksum produced uses modulus 65536 arithmetic.

Function: False
 Syntax: int GEN_False(void);
 Arguments: none
 Return Value: FALSE
 Remarks: Returns the HAL value of FALSE.

Function: True
 Syntax: int GEN_True(void);
 Arguments: none
 Return Value: TRUE
 Remarks: Returns the HAL value of TRUE.

Function: BlkEqu (#1)
 Syntax: void GEN_BlkEqu(char __far *Blk1, char __far *Blk2, int Qty);
 Arguments: Blk1 - Pointer to start of first memory block
 Blk2 - Pointer to start of second memory block
 Qty - Size of memory blocks
 Return Value: TRUE if the two memory blocks are identical
 Remarks: Carries out a byte-for-byte comparison of two memory blocks.

Module : CRC Generation

The CRC Generation module provides an application with the ability to calculate the CRC over a specified memory block. The algorithm used is intended to be compatible with CCITT and XMODEM requirements; i.e., it uses the formula :

$$X^{16}+X^{12}+X^5+1$$

The preload value should be 1 for CCITT and 0 for XMODEM use.

Function: Generate
 Syntax: int CRC_Generate(int Crc, char __far *Block, int Len);
 Arguments: Crc - Preload value
 Block - Pointer to first byte of memory block
 Len - Size of supplied memory block
 Return Value: The CRC over the supplied block
 Remarks: The routine uses a table driven algorithm and is therefore relatively fast.

Application Download Mechanism

The Kestrel terminal download application facility is achieved using an extension of the FAMS protocol. It is implemented as part of the low level operating system within the terminal that must be triggered by the currently running application. Once a download sequence has been requested, the terminal remains in it's download state until either an application has been completely downloaded or the operation has been aborted by the user.

Due to the fact that the download operation is carried out using the FAMS protocol, only limited facilities are available within a FACT system.

On each terminal power up, the terminal carries out a check to see if any downloaded application is available and, if so, it is executed in preference to the terminal's inbuilt applications. It is the responsibility of the replacement application to allow itself to be discarded; either by replacement with a new download or by returning control to the inbuilt application.

There is no supplied mechanism to allow one application to transfer data to another application - Immediately prior to running any new application for the first time, all available memory within the terminal is cleared.

When the terminal is in it's download state, it is able to recognise a limited collection of FAMS commands. These include :

'**'	-	Resync request	(Does nothing but return '**')
'A'	-	Acknowledge	(Does nothing but return 'A')
'r'	-	Retransmit	(Retransmits last response)
'@'	-	Download	(Download application)

All unrecognised commands are ignored by the terminal.

The download command (character '@') is an addition to the FAMS protocol that is used to carry out the complete download sequence. Individual steps of a download sequence are specified using a subcommand character. The subcommands supported include :

'!	-	Prepare	(Trigger download sequence)
'V'	-	Version	(Return API version details)
'?'	-	Statistics	(Return memory statistics)
'.'	-	Abort	(Return to inbuilt application)
'>'	-	Block load	(Load a single application block)
'R'	-	Application run	(Run a downloaded application)

All unrecognised subcommands (including a missing subcommand) cause the terminal to reply with a standard response of 'F'.

In general, a download subcommand that has an illegal form or parameter causes it to return a 'download fail' response. This, as will be seen, is different to the standard FAMS style 'F' response.

All download related commands and responses have been designed to allow for future expansion. To this aim, responses are of a slightly different layout to that of a normal FAMS message; but they do still follow the same basic structure.

Application Download Sequence

Details of the download commands and responses will be given in the order in which a typical host application would issue them during a download sequence.

1) Prepare '@![Parameters]'

Initially, the terminal will be running an application that must self terminate by instigating a download sequence. This subcommand must be fully interpreted by the application any is partially interpreted by the download routines.

The FAMS application, currently, allows for an optional list of communication parameters. If any parameter needs to be specified, then the complete list must be given. Additionally, the terminal will reject any request if it has any un-released transaction data pending.

The parameter list may contain (in the following order) :

Identity:1	-	Terminal identity in the range 0x00..0xFD.
Interface:1	-	Terminal interface: '0'==RS232, '1'==RS485.
Baud:1	-	Comms rate: '0'==600 Baud..'6'==38400 Baud.
RxTout:1	-	Receive timeout multiplier (in 50mS steps).
TxDelay:1	-	Transmit delay multiplier (in 1mS steps).
TxTerm:1	-	Transmission termination: '0'==NO,'1'==YES.

The values 'RxTout' and 'TxDelay' are single byte binary values that should be limited to the range 0x00..0xFD; i.e., they must not interfere with normal FAMS message requirements.

If no parameters are given, then the terminal will assume that it's current operational parameters are to be used.

Note that once the terminal is in it's download state, any parameters given as part of a subsequent '@!' command are ignored.

The inbuilt FAMS terminal application, and a compliant terminal application, will respond with a message of the form :

```
Fail: (Addr)@<Cr>!F(Reserved)<0xFF>(Check)
Ack: (Addr)@<Cr>!A(Reserved)<0xFF>(Check)
```

An application, on accepting this command, automatically requests the download state and then self terminates.

If the terminal is already in it's download state on receipt of this command, then it assumes a restart is imminent and discards any previously received data blocks.

2) Version '@V'

This command may be used by an application download routine to determine the revision level of the terminal's low level code so that it can decide what facilities are supported.

The terminal responds with a message of the form :

```
Fail: (Addr)@<Cr>VF(Reserved)<0xFF>(Check)
Ack: (Addr)@<Cr>VA**-***-***<Cr>(Reserved)<0xFF>(Check)
```

3) Statistics '@?'

Prior to an application download, the host may want to ensure that enough memory is available within the terminal to hold the application - And to subsequently run it.

The terminal responds with a message of the form :

```
Fail: (Addr)@<Cr>?F(Reserved)<0xFF>(Check)
Ack: <Addr>@<Cr>?A(Blk1Beg:5):(Blk1Len:5)<Cr>
      (Blk2Beg:5):(Blk2Len:5)<Cr>
      (Reserved)<0xFF>
```

The values 'Blk1Beg' and 'Blk2Beg' specify the starting point (relative to the start of it's one megabyte boundary) of the memory block. Both are supplied as five ASCII Hex characters.

The values 'Blk1Len' and 'Blk2Len' specify the amount of memory available within it's one megabyte boundary. Both are supplied as five ASCII Hex characters.

If no memory is available within a block then the 'Beg' and 'Len' values would be "00000".

4) Abort '@-'

The host may, at any point during an application download sequence, decide to abort the download sequence by issuing this subcommand - At which point the terminal resets itself and runs one of it's inbuilt applications.

The terminal responds with a message of the form :

```
Fail: (Addr)@<Cr>-F(Reserved)<0xFF>(Check)
Ack: (Addr)@<Cr>-A(Reserved)<0xFF>(Check)
```

5) Block load '@>[Parameters[Data]]'

The terminal receives an application as a series of data blocks. These blocks must be sent sequentially by the host and are acknowledged individually by the terminal.

As blocks of data are received by the terminal they are stored to a specified memory area within the terminal - With one block being stored immediately after it's preceding one.

To ensure that the transfer remains synchronised, each data block contains a sequence number that is generated by the host and, in turn, checked by both terminal and host.

The first data block within a download sequence has a sequence number of zero. Each subsequent data block has a sequence number greater than that of it's previous one - If the terminal receives a block that contains a sequence number less than the one expected then the data within that block is ignored.

One exception to the normal sequence number checks carried out by the terminal also exists - If the received sequence number is found to be zero, then the terminal takes this as being a request to restart the application download. All previously received data blocks are discarded and the data block associated with the message is stored at the start of the specified memory area.

The parameter list may contain (in the following order) :

SeqNum:5	-	Block sequence number (ASCII numeric).
BlkId:1	-	Destination block: '0'=1st Meg, '1'=2nd Meg.
Size:4	-	Size of following data block (ASCII numeric).

If the 'Size' is non zero then a data block (of the stated size) should follow the FAMS command header.

If no parameter list is given in the command, then the terminal assumes that the host just requires a response - An action that can be of use in communication error recovery.

The terminal responds with a message of the form :

```
Fail: (Addr)@<Cr>>F(Seq:5)<Cr>
      (Blk1Str:5):(Blk1Spc:5)<Cr>
      (Blk2Str:5):(Blk2Spc:5)<Cr>
      (Reserved)<0xFF>(Check)
Ack: (Addr)@<Cr>>A(Seq:5)<Cr>
      (Blk1Str:5):(Blk1Spc:5)<Cr>
      (Blk2Str:5):(Blk2Spc:5)<Cr>
      (Reserved)<0xFF>(Check)
```

The value 'Seq' specifies the sequence number of the data block that was last received and successfully processed by the terminal. If the terminal has yet to receive a data block, then the value returned in this field is "*****".

The values 'Blk1Str' and 'Blk2Str' specify the point (relative to the start of it's one megabyte boundary) at which the next received data block would be stored. Both are supplied as five ASCII Hex characters.

The values 'Blk1Spc' and 'Blk2Spc' specify the space available for further data blocks within its one megabyte boundary. Both are supplied as five ASCII Hex characters.

6) Application run '@R[Parameters]

Once all the application has been downloaded, the terminal must be told to terminate it's download state and run the application received.

The parameter list may contain (in the following order) :

RunChk:1 - Carry out 'pre-run check': '0'=NO, '1'=YES.

If the parameter 'RunChk' is not supplied, then it is assumed that the option is required.

The terminal responds with a message of the form :

Fail: (Addr)@<Cr>RF(Reserved)<0xFF>(Check)
Ack: (Addr)@<Cr>RA(Reserved)<0xFF>(Check)

Application Execution

Immediately prior to the execution of a downloaded application (i.e., at each terminal power up) the terminal can optionally pause for 2500mS during which time it shows the message :

"APPLICATION VERIFICATION IN PROGRESS".

If any key is pressed five times in rapid succession during this time then the application (and any data maintained by it) is totally discarded - The terminal then attempts to execute one of it's inbuilt applications.

This 'discard' facility may be of particular use during application development, where a program may not (for whatever reason) allow normal termination. The facility can be disabled by setting the 'RunChk' parameter to '0'.

Download Instigation

Manual entry to the terminal's download state is possible on both the FAMS and FACT inbuilt applications. On the shop floor applications this is achieved using diagnostic 35 and on time and attendance applications it is achieved via 'SETUP.TEST.DOWNLOAD'. Any download request is rejected by the inbuilt applications if transaction data is pending host acknowledgement.

When the download is triggered via a FAMS application, the terminal's operational parameters are selected for use during the download sequence; however, if the terminal does not have a valid identity then a value of 1 is assumed.

When the download is triggered via a FACT application, the terminal elects to use a selection of default parameters during the download sequence. These defaults are :

Terminal identity:	1
Interface:	RS232
Communication rate:	9600 Baud
Receive Timeout:	50mS
Transmit delay:	0mS
Transmit terminate:	Disabled

If the terminal does not have any inbuilt applications, then it has no choice but to enter the download state immediately. In this situation it elects to use the same default parameters as of those used by a FACT application.

Once the terminal has entered it's download state, it normally remains there until an application has been received - If power is removed and then returns then the terminal still stays in it's download state.

The download state can be prematurely terminated by a suitable communication command ('@-') or by pressing any terminal key five times in rapid succession during a 2500mS period.