Application Note AN2002 Description of the Application Programming Interface of the HA40

Exported on 08.10.2021

pei tel Communications GmbH
info@peitel.com
www.peitel.com

Table of Contents

Redactions hints:

- Remove/replace the bad fixed section/table numbering
- Check for denglish
- Check internal broken references

History

Date	Revision	Author	Comments
12.11.2020	1.0	RaSc	First Release
31.03.2021	1.1	RaSc	Clarification for protocol expansions, old pictures replaced by correct text

Table 1: Document History

Related Documents

No.	Name	Remarks
1	AN1800 HA40 User Manual	Application Note 1800
2	AN1903 AppLoader HA5x SDK & HA57 EVO	Application Note 1903
3	AN1904 LogoLoader for Handsets	Application Note 1904

Table 2: Related Documents

Introduction

This document describes the usage of the Software Development Kits (SDK) for the handsets of the HA57/HA40 Series by pei tel Communications GmbH with API Version 4. It describes the structure and the handling of the SDK in detail. Using this documentation will enable the user to create his own user applications for the HA40, and enhance or replace the usable fonts. In rare maintenance cases also the command protocol can be modified.

Note: for historical reasons and for compatibility with HA57 EVO projects the old name scheme (HA57evo prefix) of example projects as well as the names of interfaces are not changed.

Please ensure that you have the documents listed in Related Documents available for reference.

Glossary

Application	Software to be loaded on the HA40 using the provided firmware as the operating system.
Firmware	The system software of the HA40. It provides all the necessary functions and procedures to access the resources of the HA40.

Abbreviations

API	Application Programming Interface
RAM	Random Access Memory
SDK	Software Development Kit

Prerequisites

The SDK is available on all HA57/HA40 handsets. The development of user applications requires a development system for the ARM Cortex M4 processor. Two versions were tested thoroughly and can be recommended:

1. µVision V5.28 (KEIL™ Software) is a commercial solution used for the firmware
2. Atollic TrueSTUDIO® for STM32 in combination with the GNU-C-Compiler (ARM-GCC) is a freely available solution

Software Architecture

The firmware of the HA40 is a thread-based system. The access to the hardware resources must be conducted by driver functions. An uploaded application must be recognized by the system. It is started as a task and gets access to the system functions provided by the firmware. A direct access to the hardware components for the HA40 is not allowed. Due to the CPU architecture, there is no enhanced protection from illegal accesses.

If an application creates a detectable hardware error (FAULT) or a busy loop (watchdog), the device restarts automatically.

An application is constructed around an event handler function. This handler receives events from the system and processes them by triggering appropriate reactions. The API is realized in form of a symbol table handed over to the application.

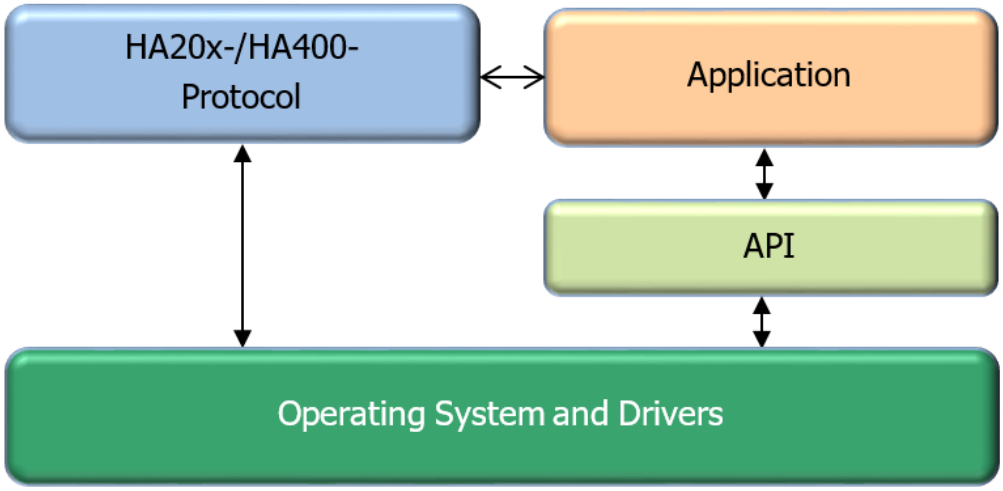

Figure 1: Software Stack

The SDK API allows:

- SDK API applications: the user code replaces the internally implemented protocol handler and has to implement its own serial communication with arbitrary serial settings. The approach with most freedom and most work.
- SDK API protocol expansion: the user code can inject actions before protocol command execution, i.e. the configured protocol handler is running. It is used for problem fixing and command extensions. Important note: in the case of protocol support requests, the expansion code has to be provided to pei tel development.
- SDK API font extension: The font injection is independent from expansions or applications and allows the enhancement of the available UTF set of the HA40. It can be used for fast icon implementations (recommended) or complete UTF page integration (demonstrated with Greek). Note: font images are large, pei el development can alternatively create font updates located within the serial configuration flash instead of the CPU program flash.

Components of the SDK

The SDK consists of this documentation, an application programming tool, project templates for the KEIL™ and GCC tool chains and demo applications.

The development environment can be chosen by the user taking into account the necessary requirements. One of the project templates may be modified accordingly.

For compatibility reasons the interface prefix "HA57evo" is reused for HA40. All use of "HA57evo" can also be seen as "HA40". The main target is compilation compatibility to HA57evo. Binary compatibility is not possible because of different processor types.

Reference Applications

A project file with a project template as well as a detailed example application for both application methods of the SDK are provided for both of the common development environments, KEIL™ and the STM processor free development environment, Atollic TrueSTUDIO® based on gcc.

The example highlights the programming approach and the handling of the API in detail. The project template can be used as a basis for own developments. This approach needs to be modified for different development environments accordingly to the IDE in use.

Empty project templates are:

- AppHA40EmptyProject: template for application
- ExpHA40EmptyProject: tempalte for protocol expansion

Example Projects:

- AppHA40SerialProject: test application to check serial connections with an RX-TX loop connector, good example for serials readings
- AppHA40TestSDK: our half automated SDK API test, uses nearly all API functions and could help understanding the API
- FontExpansion: demonstrates the font expansion by adding Greek letters
- ExpHA40TestHA20X: small HA20X command modification example
- ExpHA40TestHA400: small HA400 command modification example

KEIL™ and Atollic

Example and empty template projects contain supports both project structures: Keil and Atollic.

Components of the Project Template

The project template consists of:

- A project file with guidelines for the development environment
- A folder Source/: 2 c files (entry point and application)
- A folder Include/: 1 h file (application)
- A folder System/: 1 h file (API definition), 1 assembler start-up file, (the ld file is not required for the µVision compilation)

Reference Examples

The reference project for KEIL™ contains:

- System/
  - HA40_startup.s: start-up code for Keil only
  - HA40Mem,ld: linker script for Atollic only
  - HA57evo_system.h: SDK API
  - HA57evo_protocol_ext.h: SDK API for protocol extensions (optionally)
- Source/ (selections depends on the project type)
  - App_main.c or ExpEVO_main.c: template for applications or protocol expansions
  - App_prog.c or ExpEVO_prog.c: empty program template
- XXX.uvprojx: Keil project file
- .project, .cproject: Atollic project structures

Suggested Programming Approach: Keil

For the implementation of a custom application using the KEIL™ development environment, the following approach is suggested:

1. Copy the project template to the working area.
2. Start the KEIL™ development environment.
3. Open your copy of the project in the development environment.
4. Rename the application files as desired.
5. Compilation. Verify that the compilation runs without errors and warnings!
6. Start to add your own application functionality.

Suggested Programming Approach: Atollic

For the implementation of a custom application using the "Atollic" development suite, the following approach is suggested:

1. Copy the project template to the working area.
2. Start Atollic (Eclipse based).
3. Import/Open your copy of the project in the project explorer.
4. Select the root directory.
5. Compilation. Verify that the compilation runs without errors and warnings!
6. Start to write your own application.

Alternative Compilation

The following presents a collection of technical details for an alternative compilation, e.g. a cross compilation in Linux with an ARM-GCC or the new CUBE IDE It is necessary to follow these steps in order to create the hex files needed for the application:

1. Compilation of the c files to objects. Use the relevant include paths of the local implementation (Include/), the SDK API (System/) and the standard Lib for necessary type definitions (that's where stdint.h can be located). The firmware compiles ARM Thumb, but any desired combination of the generation should work as well.
2. Linking is done without standard libraries and the entry point "application" of the main c file. This function is located at the address 0x08060000 (in thumb mode this might be shown at 0x08060001). The firmware jumps to this address, with the symbol table being the first argument when the application is active. There is no initialization like, e.g. a library, which means that global variables/objects of the application code can't be initialized. Therefore, it is not recommended to use C++ or custom-made libraries. Any dynamic loader functionality needs to be integrated in "application". For the GCC code, there are demonstrated concepts, but they extremely depend on the compiler.
    a. For KEIL™, an own .entry area (section) is used for this functionality, which is accordingly placed first in the asm file.
    b. The Atollic project uses a linker script, which moves the respective .entry area to the necessary address.
3. The created binary needs to be transferred into the Intel Hex format, or respectively, some linkers are able to create the format directly. If correctly linked, this file should only contain addresses of the flash area 0x0806000 to 0x8100000. RAM areas should be without code.

The above-mentioned sections are filled with code during compilation according to certain rules. If static code is not relocatable, the linker will assign real addresses to the sections. For dynamic libraries, there is relocatable code. This assignment (and as well the dereferencing of references to other areas) occurs during runtime using a dynamic linker.

We explicitly do not use this; therefore, a non-relocatable, statically linked code needs to be created (option of the compiler/linker).

Common sections are:

- .code/.text: The program code in a read-only section. Sub-sections are possible, which contain fixed stings in the code.
- .rodata: All global/static **constant** objects. These remain in the directly addressable flash.
- .data: The values of all global/static objects placed in the flash which are not pre-initialized with 0 and which are at start-up copied from this section into the RAM area. The linker script of the gcc delivers appropriate conditions, so that the "memcpy" in "application" can do the copying. If this copying does not occur (KEIL™), all global variables will not be initialized.
- .bss: A 0-initialized RAM area for 0-initialized objects (these are practically all static, non-initialized variables/objects). GCC will set this area via memset to 0, KEIL™ is missing this step. Therefore, all static/global variables/objects remain uninitialized.
- .entry: The code area of the function "application" with compiler dependent indication of a section in the C code. This area needs to be set to the address 0x08060000.

Notes

- Test the correct functioning of the printf variants with the variable parameters. At this point, different compilers will produce discrepancies (stack direction and endianness are relevant!).
- Variants of vprintf can have other restrictions (limited stack usage).
- In case the compiler removes unused code, mark the "application" function as "used". Otherwise, the created binary will soon be empty.

AppLoader HA57 SDK & HA57 EVO

The program "**AppLoader HA57 SDK & HA57 EVO**" is a Windows® application that provides the means for uploading and starting a user application also on the HA40 handset. The upload procedure requires a free serial interface to be available on the PC.

Please find a detailed description of the uploader program in the documentation "AN1903 AppLoader HA5x SDK & HA57 EVO". The most actual version also supports HA40.

> **Note**
>
> Before loading an application with a standard (115200 Baud) baud rate, the internal emulation "pei tel HA20x" or "pei tel HA400" must be activated or the original HA40 system menu must be open, otherwise the handset will not be recognized by the AppLoader tool. See also document "AN1800 HA57 EVO User Manual", section 4.4 Emulation.

Porting

From HA57 SDK

Features and processes are based on the HA57 SDK layout (HA57 SDK: predecessor model). Special attention was paid to the compatibility of the features. Due to new hardware components and technical progress, some changes occurred, therefore, please pay attention when using already available software:

- The display's resolution increased to 240x320 and the colour space to RGB-565. Therefore, absolute coordinates (not relative to display size) need to be doubled in most cases. Colour specifications not based on constants need to be converted. The macro RGB565_COLOR gives a hint how to do this.
- The SDK API doesn't rely any more on the fixed addresses of the firmware. Instead, a symbol table is handed over to the entry function of the application. This changes initial procedure; the central event handler is no longer the entry. It is set, or not set, in the structure appl_Info_s together with the font handler. Setting the object appl_Info_s needs to be complete, in accordance with the type definition. It is necessary to match the main file with the new AppEVO_main.c file. There's an advantage in this: After a change in the firmware, the application doesn't need to be recompiled. The entry in the API function is respectively *table->function*. Common function calls are provided by C macros.
- Most of the text functions use now UTF8 instead of the mostly not documented ASCII coding (this isn't valid for text-mode based functions). We assume this will be compatible to most of the usages so far.
- The flash API is missing, as there isn't an external flash hardware for the HA57 EVO. Querying the flash values, will return, as the HA57 without flash did, a 0.
- The HA57 SDK implementation of the cursor in *appl_Info_s* gives non-explicable results. Therefore, this field will be ignored. However, it will remain part of the structure.
- The font names contain the normal/bold properties, therefore, the parameters *appl_letter_size_t* are not needed. However, if the value "bold" is set, a normal font will be displayed in its bold version, if available.
- The function *appl_LCD_setTextOffset* gives, when using non-standard fonts, inexplicable results. Using it creates a risk that elements of the graphic areas overwrite each other.
- Images (bmp) are normally filed in form of a byte array in the C code, adjusted to display resolution and colour coding. Both have now changed. To perform a fast translation, put a bmp into the brackets of the function *appl_LCD_setCompatibility*. Then scale the image by factor 2 and adjust the colours to RGB-555. Nevertheless, the coordinates need to be adjusted accordingly.

  > **Note**
  >
  > The memory space needed for double resolution quadruples!

- Bitmaps were offered as a replacement for pictograms or missing fonts. The bitmaps in the protocols are compatible, meaning, scaled by factor 2. The SDK offers the scaling as well. The function *appl_LCD_setCompatibility* can be used for scaling. Otherwise, you can just create new bitmaps. It is recommended, in regard to the new UTF font enhancements, to do without.
- The definition for *appl_hw_description_s* has been changed, as the HA57 EVO has a PPT button, but not unbalanced audio. In addition, the serial interface can only be used as RS232-V.24.

From HA57 EVO

- The display of HA40 is rotated, this heavily changes the available pixel space and absolute coordinates should be adapted. Now, most graphic functions implement a clipping behaviour instead of complete fail with out-of-range parameters.

- Icon and navigation area are much smaller (one normal font line height) than with HA57 EVO to guaranty 8 text lines within the text field.
- HA57 EVO fills a line with 16 characters monospace normal font. HA40 allows 20 characters, the regular text mode (8 lines 16 characters) uses a left/right margin.
- If the new power off event is answered as proposed in the templates, the device remains on (as an HA57 without on/off support). With default settings, there is no on/off support activated, so nothing changes.
- If the new asynchronous event is ignored, there is not ambient light support, no distance effects and no headset support, exactly as with HA57 EVO.

Application Programming Interface (API)

The API functions provide the access to the resources of the HA40. The available functions are declared in the "HA57evo_system.h" header file. This file also contains the most actual documentation in English. Especially, the sample applications for the API tests are extensive applications with all available functions and can be used as additional documentation.

As all functions are conveyed with a symbol table, there is a set of macros, which offer the functions analogue to the previous SDK. Due to the symbol tables, the application is independent of the firmware version. The firmware can increase to the API version, which will never change available symbols, but might add new ones. Note: HA40 uses a different processor, there is no binary compatibility to HA57 EVO or HA57 EVO SDK.

Memory Map

The memory areas available for the use by applications are marked in **GREEN**.

System (reserved)	0xFFFF FFFF
	0x2004 0000
RAM (Application)	0x2003 FFFF
	0x2001 0000
RAM (Firmware)	0x2000 FFFF
	0x2000 0000
System (reserved)	0x1FFF FFFF
	0x0801 0000
Flash (Application)	0x080F FFFF
	0x0806 0000
Firmware	0x0805 FFFF
	0x0800 C000
Bootloader	0x0800 BFFF
	0x0800 0000
System (reserved)	0x07FF FFFF
	0x0000 0000

Figure 6: Memory Map

The memory offers 192 kB RAM for static and global objects or data as well as 576 kB FLASH for code and constant data (fonts, icons, images).

> **Note**
>
> Stack and Heap (malloc) are part of the firmware, which limits it to the 64 kB area! The usage of many or large local variables or recursions can lead to a stack overflow.

Entry Point

The application entry point is fixed to the absolute address **0x0806 0000**.

This address is used as function entry point to hand over the symbol table and to receive the description of the application in exchange back. After the one-time entry, all other referencing occurs on basis of the handed over objects.

Symbol Table

The structure sdk_symbols_s consists of a version field followed by pointers to functions. Those pointers represent all the available entry points in the direction of the firmware. The handed over object is stored in the flash and therefore can't be modified. A set of macros is available in order to avoid that functions need to be called in a complex way using symbol table objects. The current solution is compatible to the HA57 SDK/EVO.

> **Note**
>
> Symbol tables type-safely implement all function calls in the firmware as well as in the application. Attention should be paid to any warnings issued by the compiler.

After a firmware update on the device, the SDK version might increase and functions might have been added at the end of the structure. This doesn't matter for older implementations.

Versioning

The version concept with an SDK is more complex than a simple firmware of a device. Firmware and Bootloader have their own versions, which in most cases consists of a main version and a subversion number, even in some cases of a build number or a patch level. These versions are independent of the SDK. The version of the SDK relates to an API version, which ideally shouldn't be changed with the firmware. SDK expansions are allowed, which changes the subversion of the SDK, but not the firmware (only the build number, as firmware functions should remain unchanged). Therefore, SDK versions and firmware versions are independent values.

In addition, there is an SDK version from the point of view of the writer of an application, which is noted in the software (which is installed on the PC) for the SDK (in HA57evo_system.h). This is the version of the used SDK API. The device firmware, where the compilation is loaded to, can offer a different SDK version (existing SDK version). There is no problem if the subversion of this firmware is higher, but, if it is lower, the registration of the application won't work. The entry function sees the current SDK version of the firmware in the system table and needs to write the required version of the compilation into the user info.

Required Code Skeleton

A specific code skeleton is required to create an application. Compliance with this scheme guarantees a correct integration of the application into the running system and its registration in the Setup for correct (re-/de-) activation. The registration process in the Setup is described in detail in the user manual.

Type Name Definition

The following type names were predefined to the application type:

appl_types_t	typedef enum

appl_types_t

This data type provides the application type for the registration.

Syntax

```
typedef enum
{   ATYPE_NON,
    ATYPE_PROGRAM,
    ATYPE_RESERVED_1,
    ATYPE_RESERVED_2,
    ATYPE_FONT_EXTENSION
}   appl_types_t;
```

Parameter

```
ATYPE_NONE
    No application is activated, meaning that there is no jump to the callbacks of the
application.

ATYPE_PROGRAM
    The application controls the complete HA40 using the API and replaces the protocol
handlers of the serial interface.
    Therefore, loading of applications is not possible for this activation.
    The serial interface works byte-oriented without a command interpreter, binary
transfers should work without problems.

ATYPE_RESERVED_1, ATYPE_RESERVED_2
    The application complements or modifies the HA20x/HA400 protocol. Some API areas
of the PROGRAM mode are not available.
    The serial interface works command-oriented. Using that kind of application is
restricted and requires the protocol interface
    file.

ATYPE_HA20XEXT (protocol extension only)
The application complements or modifies the HA20x protocol. Some API areas of the
PROGRAM mode are not available.
The serial interface works command oriented.

ATYPE_HA400EXT (protocol extension only)
The application complements or modifies the HA400 protocol. Some API areas of the
PROGRAM mode are not available.
The serial interface works command oriented.

ATYPE_FONT_EXTENSION
    The application only complements or modifies the fonts of the HA40. The event
callback is not used here.
```

Struct appl_Info_s

This structure is used to report

- the name of the application
- the SDK version at compilation (can deviate from the firmware version after an update)
- the type of the application
- the event callback
- the font handler

to the system. When, at activation, the SDK version is higher than the version of the current system or even wrong (magic code), the message "**Application Version Mismatch**" is shown on the display for 10 seconds. In this case, the application is not activated.

Both callback functions are allowed to be NULL, but it does not make sense to load an application without callback function.

The application type indicates the way, how the code of the application is integrated. An application of the type "PROGRAM" takes over total control of the handset after boot-up. A handset of the type "EXTENSION" extends the functionality (e.g. fonts) of one of the protocols of the HA40.

Definition

```
typedef struct
{   const char *applName;
    #define SDK_MAGIC 0x06022019

    #define SDK_IF_VERSION 4
    #define SDK_VERSION (SDK_MAGIC + SDK_IF_VERSION)
    uint32_t SDK_VersionCode;
    appl_types_t applType;
    void* (*application)(unsigned short event);
    void (*char_info)(uint32_t utf32, appl_typeface_t font, appl_char_info_t *info);
}   appl_Info_s;
```

Parameter

```
applName
    Name of the application for menu selection

SDK_MAGIC
    Magic code for structural testing, can also be seen as major version identifier

SDK_IF_VERSION 4
    Requested SDK interface version. Might be lower than the implemented version in
the device.

SDK_VERSION(...)
    Value to be inserted into the field SDK_VERSION code.

SDK_VersionCode
    Handed over value for version testing.

applType
    Requested type of application

void* (*application)(unsigned short event)
    Applications event handler (function pointer) with the argument:
    event: Mask, dependent on the type of application.
    The return type is either NULL, which makes the device reboot in the ATYPE_PROGRAM
mode or a pointer (which might be NULL as well) to a status value for both protocol
extensions.

void (*char_info)(uint32_t utf32, appl_typeface_t font, appl_char_info_t *info)
    Font handler (function pointer) with the arguments:
    utf32           the UTF32 character to be drawn
    font            the font to be used (size and formatting)
    info (in/out)   character object to be pre-filled and continuously refilled.
Expected font, width, height, no bitmap, character width 0.
```

The application function needs, preferably, as return a static object of this structure. It can be stored in the flash constantly as shown in the examples.

Note

This structure was modified slightly compared with the old HA57 SDK and the handover was changed.

System Events

The program sequence of an application is controlled by events. Therefore, each action is triggered by an event. The following table shows all available events.

Only use the events that are available for the respective application type.

Event	Description
EV_KEY_INPUT	A key was pressed.
EV_KEY_RELEASED	A key was released.
EV_KEY_TIMEOUT	The KEY timer has expired.
EV_HOOK_INPUT	The state of the HOOK switch changed.
EV_PTT_INPUT	The state of the PTT key changed.
EV_POWER_ON	The event is sent immediately after POWER ON or if the device leaves a service mode; the application software might react with an initialization of the system (display, configuration).
EV_LOGO_TIMEOUT	The LOGO timer has expired.
EV_BYTE_INPUT	A byte was received at the UART (for programs only).
EV_ASYNC	new on version 4: other hardware depended event notification (headset, ambient light, approximation,...)
EV_LIGHT_TIMEOUT	The LIGHT timer has expired.
EV_USER_TIMER1	The custom timer 1 has expired.
EV_USER_TIMER2	The custom timer 2 has expired.
EV_USER_TIMER3	The custom timer 3 has expired.
EV_VERSION_GET	Event to request the application version for the information screen (system menu).
EV_POWER_OFF	new version 4: power off request (if supported and activated), can be rejected by application

Table 3: Event List for Application Type "Program"

Function application()

This function is the entry point of the application. The application programmer must ensure that the linker places the implementation of this function at the defined entry point address 0x08060000.

The example applications and the project templates for new projects already contain the necessary linker settings.

KEIL™:

"-entry=application" is added to the linker options. The memory mapping must be defined in the Options.

GCC:

The memory map is defined by respective records in the file HA57EvoMem.ld. These settings will be loaded by the Makefile via LDSCRIPT.

Event Handler

The code of the *default* path of the event selection needs to be kept as described.

Example: Application of the type "Program"

```
appl_Info_s  applInfo;
unsigned app_version = 1;

void *application_events (unsigned short event)
{
    static int rv = 1;
    void * ptr = &rv;
    if (event & EV_LOGO_TIMEOUT)
    {   /* Timer overflow LOGO_TIMER */
    }
    if (event & EV_KEY_TIMEOUT)
    {   /* Timer overflow KEY_TIMER */
    }
    ...
    if (event & EV_VERSION_GET)
    {   /* Application Version request, only used for the production process
documentation on preinstalled applications. */
        ptr = "1.00.01";    // any address in FW flash is interpreted a 0 terminated
string (max size 12)
        ptr = &app_version;  // else if not NULL, the result is interpreted as
unsigned int pointer for a version with one number
    }
    if (event & EV_POWER_ON)
    {   /* First call after power on or after leaving a service mode, can be called
multiple on one power cycle! */
        // applicationInit();
    }
    if (event & EV_POWER_OFF)
    {
        return NULL; // This was the last call, the device switched off
        return ptr;  // Power off request is rejected but can be initiated later by
API, Note: emergency power off cannot be rejected!
    }
    return ptr;
}
```

The return value of the function "application" is a pointer. This pointer is interpreted differently depending on the type of application.

Program:	Pointer to requested information or pointer to 1 (mostly the value is ignored) In case of a *NULL* – abortion of the application and restart (possibly switch off the device).
Expansion:	For events that signal an action, the pointer points to 0, when the signalled input was treated conclusively. When the pointer points to 1, the input is still treated. For events that request information, the pointer points to the required information.

Font Handler

The font handler universally replaces the bitmap functions of the old SDK on UTF basis. The handler gets an UTF32 value, the font and a prefilled char structure which contains the expected height and width of the font in pixels. For particular characters, the values might differ. The reference for the placement on the display is always the upper left corner of the character. It is expected that the bitmap of the character and resolution of the bitmap are registered. If not, the compiled or SPI flash font descriptions are used or a replacement character (blank space) is displayed.

In this way, fonts of the HA40 can be extended or replaced.

Definition

```
typedef struct
{   uint8_t font_width;
    uint8_t font_height;
    uint8_t height;
    uint8_t width;
    const uint8_t* bitmap;
    uint8_t alias;  // new since API version 4, preset with 1 for compatibility
}   appl_char_info_t;
```

Parameter

```
uint8_t font_width;
    Font width (for monospace fonts and spaces inclusive blanks), prefilled

uint8_t font_height;
    Font height, not to be confused with pixel height, prefilled

uint8_t height; prefilled with 0
    Height in pixel

uint8_t width;
    Prefilled with 0 (no answer from application)
    Width of the bitmap in pixel

const uint8_t* bitmap;
    Size of the given bitmap should be:
    height * ((width + 7) & ~7) * alias / 8 bytes
    If NULL a blank is displayed (with the correct width).


uint8_t alias;
    Number of bits per pixel field, normally 1
    also allowed are 2 (4 colors), or 4 (16 colors) to improve the readability for
 characters
    other values are handled as 1.
```

The example **Font extensions** shows the default UTF8 implementation currently available in the firmware.

In order to consider memory space, doubling the alias value doubles also the required space for bit maps. It is possible to preload alternative font sets to SPI flash, contact your distributor for desires.

Special Features

The HA40 API offers no virtual or especially protected environment. The code runs directly in the context of the firmware and especially without the normally usual C start-up. Here are the consequences:

- Crashes will also crash the firmware, which will, in most cases, be able to react according to the fault handler. When the fault handler locates the crash, the device restarts.
- The idle handler of the system offers a simple busy loop detection. When the event handler needs too much time, the device will be rebooted by Watchdog. A delay function (as well with 0) as argument triggers the Watchdog. Generally, it is recommended to avoid busy loops in the code (see also coding of UART tests).
- When linking own application with own libraries, the required start-up initializations are not called. This is also valid for the 0-initialization of static/global variables, pre-initialized objects and especially the necessary C++ start-ups (ctor) at C++ usage. Therefore, such code won't work at some places. With special effort, it is possible to implement this at the entry point of the application for the used compiler. However, this is not part of the SDK API! For GCC there is an approach to realize the 0 initialization and pre-initialization of the objects (might not work depending on the compiler). In general, it should be assumed that all variables are not initialized!
- Event functions are called by exactly one thread of the firmware and do not need to be implemented reentrant. It also rare, that more than one event is set (for compatibility reasons this is not a hard statement).

Interrupt events (timer, serial data, keys…) are handled per event and respective firmware threads and might be delayed.

- If the application crashes directly on start, there are two ways to break the restart cycle. First call the Bootloader using the key combination "red + green + star" on boot. The load screen appears. Now
  - load a new application version with the application loader tool or
  - the system menu can be called from pressing star + green in that sequence. Here the menu starts without calling the application start-up (which can deactivate the system menu) and the application can be deactivated. It is also possible to use the application loader with the active system menu to replace the defect application. If the application crashes if the application name is read for the Emulation menu entry, the load-flag of the application is removed and the application entry disappears with the next call.

Display

This section describes all the functions provided for writing to the display. These functions range from the display of plain text all the way to controlling individual pixels of the display.

Defined Types

The following type has been defined for the use by DISPLAY functions:

appl_display_mode_t	typedef enum

appl_display_mode_t

For a simple display of fonts, especially for normal protocol functions, pre-defined display profiles for the text area (SECTION B) can be selected. These control implicitly the font size, placement and the mapping of bytes to UTF characters (compatible to the protocol description). Of course, current developments should be based on UTF. When the text functions are used for display modes, the firmware provides a text memory and the necessary refresh of the text is reduced to changes.

Syntax

```
typedef enum
{   DISP_MODE_ASCII = 0,
    DISP_MODE_SMS = 1,
    DISP_MODE_TB = 2,
    DISP_MODE_CYRILL = 3,
    DISP_MODE_BIG_SHIFT = 4,
    DISP_MODE_BIG_CENTER = 13,
    DISP_MODE_BIG_CENTER2 = 14,
    DISP_MODE_UTF8 = 17
}   appl_display_mode_t;
```

Parameter

```
DISP_MODE_ASCII
    ASCII mapping is active, text display 8 lines 16 characters

DISP_MODE_SMS
    SMS mapping is active, text display 8 lines 16 characters

DISP_MODE_TB
    TB (telephone book) mapping is active, text display 8 lines 16 characters

DISP_MODE_CYRILL
    Cyrillic mapping is active, text display 8 lines 16 characters

DISP_MODE_BIG_SHIFT
    Mixed mode one small one big line, text starts big and is shifted left to the
small line above the big line (number typing)

DISP_MODE_BIG_CENTER
     Centered text with big lines

DISPL_MODE_BIG_CENTER2
    One big centered line and two small centered lines (used to display upload states)

DISP_MODE_UTF8
    Modern UTF-8 coding of the strings for normal display, text display 8 lines 16
 characters
```

appl_section_t

The HA20x defines three display sections:

- A: Symbol and icon area on top
- B: Text area
- C: Navigation area on bottom

Values of that type can be used as a mask:

Syntax

```
typedef enum
{   SECTION_A = 1,
    SECTION_B = 2,
    SECTION_C = 4,
    SECTION_AB = 3,
    SECTION_BC = 6,
    SECTION_AC = 5,
    SECTION_ABC = 7
}   appl_section_t;
```

Parameter

```
SECTION_A
    Section symbol area

SECTION_AB
    Section symbol and text area

SECTION_ABC
    All sections
```

Example

```
appl_section_t  section;
section = SECTION_C;
```

Functions

The following DISPLAY functions are available:

appl_LCD_clear()
appl_LCD_clearCommonScreen()
appl_LCD_clearSection()
appl_LCD_setDisplayMode()
appl_LCD_setBrightness()
appl_LCD_getBrightness()

appl_LCD_setCompatibility()

appl_LCD_clear()

This function clears the complete display using the specified colour. The colour can be selected from pre-defined colour values or custom values can be used (see also Encoding of colours).

Syntax

```
void appl_LCD_clear(uint16_t color);
```

Parameter

```
color
    16 Bit value in accordance with section Encoding of colours
```

Example

```
appl_LCD_clear(White);
```

appl_LCD_clearCommonScreen()

This function clears the text memory, resets the cursor to home position and stops active animations. Inverted lines, active cursors remain active. The areas for icons, text and soft keys are filled with their initialized background colours.

Syntax

```
void appl_LCD_clearCommonScreen(void);
```

Example

```
appl_LCD_clearCommonScreen();
```

appl_LCD_clearSection()

This function clears single sections separately or in combination on the HA20x-typical display. The display areas for icons, text and softkeys appear in the initialized background colours.

In contrast to appl_LCD_clearCommonScreen() neither animations are stopped nor text memory is removed.

Syntax

```
void appl_LCD_clearSection( section_t section);
```

Parameter

```
Sections to be deleted
```

Example

```
appl_LCD_clearSection( SECTION_AC );
```

appl_LCD_setDisplayMode()

This function selects the pre-defined display mode. This command resets the associated text buffer, the cursor position animations, inverse lines.

Syntax

```
void appl_LCD_setDisplayMode(appl_display_mode_t dm);
```

Parameter

```
dm
    Display mode specified as value of type appl_display_mode_t
```

Example

```
appl_LCD_setDisplayMode(DISP_MODE_BIG_SHIFT);
```

appl_LCD_setBrightness()

This command sets the display brightness: 21 levels of brightness are available. Values range from 0 to LCD_MAX_BRIGHTNESS (20). The value 0 switches the illumination off.

Syntax

```
void appl_LCD_setBrightness(const uint8_t brightness);
```

Parameter

```
brightness
    Brightness 0 ... 20
```

Example

```
appl_LCD_setBrightness ( 5 );
```

appl_LCD_getBrightness()

Query of the display brightness.

Syntax

```
uint8_t appl_LCD_getBrightness(void);
```

Reply

```
uint8_t
    Brightness  0... 20
```

Example

```
brightness = appl_LCD_getBrightness ();
```

appl_LCD_setCompatibility()

Some functions now interpret the colour space RGB555 instead of RGB565 and the bmp and bitmap functions scale by the factor 2. This helps to port already available HA57 SDK applications faster and also to avoid memory problems.

Syntax

```
Syntax:    uint8_t appl_LCD_getBrightness(void);
```

Parameter

```
Boolean
    TRUE    HA57 SDK compatibility mode
    FALSE   normal SDK functions
```

Text

This section describes the API for the output of text using pre-defined fonts.

Defined Types

The following types have been defined for the TEXT area:

appl_typeface_t	typedef enum
appl_text_section_t	typedef enum
appl_shapes_t	typedef enum
appl_text_description_s	typedef struct
appl_scr_text_s	typedef struct

appl_typeface_t

Generally, there are fonts in three sizes on offer and for small and normal fonts the bold varieties. They can be selected using this parameter.

Definition

```
typedef enum
{   FONT_SMALL
    FONT_NORMAL
    FONT_LARGE
    FONT_SMALL_BOLD
    FONT_NORMAL_BOLD
    FONT_NR
}   appl_typeface_t;
```

Parameter

```
Font

FONT_SMALL
    Small font

FONT_NORMAL
    Normal default font

FONT_LARGE
    Big font

FONT_SMALL_BOLD
    Small bold font

FONT_NORMAL_BOLD
    Normal bold font

FONT_NR
    Number of supported fonts
```

appl_text_section_t

A separate font type can be configured for each component of the defined text sections.

Syntax

```
typedef enum
{   TEXT_SECTION,
    SOFTKEY_SECTION
}   appl_text_section_t;
```

Parameter

```
TEXT_SECTION
    Text area

SOFTKEY_SECTION
    Softkey area
```

appl_shapes_t

Font display can be modified as follows:

Definition

```
typedef enum
{   SH_ORDINARY,
    SH_CURSOR,
    SH_UNDERLINE,
    SH_TRANSPARENT,
    SH_INVERS,
    SH_VARIABLE,
    SH_CLIPPING,
    SH_CENTER
}   appl_shapes_t;
```

Parameter

```
SH_ORDINARY
    Ordinary display: Text color on background color.

SH_INVERS
    Invers: Text in background colour, background in text colour

SH_TRANSPARENT
    Transparent (Text in text colour on existing background)

SH_CURSOR
    Underlined with 3 pixels thickness

SH_UNDERLINE
    Underlined with 2 pixels thickness

SH_VARIABLE
    Variable justification

SH_CLIPPING
    No automatic line break, clipping is used

SH_CENTER
    Center text, X coordinate is ignored here
```

appl_text_description_s

This struct collects different aspects of the formatted output of a text.

Syntax

```
typedef struct
{   char *text;
    uint16_t x;
    uint16_t y;
    uint8_t cursor;
    appl_shapes_t shape;
    appl_letter_size size
    appl_typeface_t font;
}   appl_text_description_s;
```

Parameters

```
text
    Pointer to the UTF8 text to be displayed. Must be zero-terminated.

x
    X-coordinate of the start of output (Range: 0 to LCD_MAX_X-1).

y
    Y-coordinate of the start of output (Range: 0 to LCD_MAX_Y-1).

cursor
    0 no action; if not 0 on that position (starting with  1) a cursor (SH_CURSOR) is
shown.

shape
    Display mode specified as value of type appl_shapes_t

size
    Compatibility; modifying factor for bold, can be set also by using font.

font
    Font type of the text in according with type appl_typeface_t.
```

appl_scr_text_s

This struct describes the properties of a text scrolling through a display line. The use is deprecated!

Syntax

```
typedef struct
{   uint8_t step;
    appl_typeface_t font;
    uint8_t line;
    const char *text;
}   appl_scr_text_s;
```

appl_scr_text_ext_s

API version 4

This struct describes the properties of a text scrolling through a display line.

Syntax

```
typedef struct
{   uint16_t step_ms;
    appl_typeface_t font;
    appl_shapes_t shape;
    uint16_t x;
    uint16_t y;
    uint16_t width;
    const char *text;
}   appl_scr_text_ext_s;
```

Parameter

```
step, step_ms
    Period of a single scroll step in units of 5ms,1ms

font
    Typeface specified as value of type appl_typeface_t

shape
    Modifier for text setting

x, y
    left upper corner of the scrolling text line

width
    visible width of the scroll line (in display pixel)
    x + width should be lower equal LCD_MAX_X

text
    Pointer to the UTF8 text to be displayed. Must be zero-terminated.
```

Functions

The following TEXT functions are provided:

appl_LCD_setTextOffset()
appl_LCD_setTextCursorPos()
appl_LCD_setTextSurface()
appl_LCD_writeText()
appl_LCD_centerText()

appl_LCD_displayDefinedChar()
appl_LCD_displayText()
appl_LCD_enableCursor()
appl_LCD_scrText()
appl_LCD_scrClear()

Note

Some text functions support the text modes with text memory, other display the text directly at the given coordinates without buffer.

appl_LCD_setTextOffset()

An offset in pixels is defined at start of a text field for text modes. This setting influences the text output with *appl_LCD_writeText()*.

The function is rare used to adapt scroll line positions in protocol extensions (custom specific fixes). Normally it is not a good idea to change the line geometry of the text field.

Note

For the HA57 SDK this parameter was defined quite vaguely. Therefore, the semantic has changed.

Syntax

```
void appl_LCD_setTextOffset(uint16_t toffs);
```

Parameter

```
toffs
    Distance of the text field from the upper edge of the display in pixels
```

Example

```
appl_LCD_setTextOffset(16);
```

For completeness the reverse function was added to SDK API 4:

Syntax

```
uint16_t appl_LCD_getTextOffset(void);
```

appl_LCD_setTextCursorPos()

This function places the cursor at the specified character position within the text section. The position is counted in characters from the start. The current cursor position affects the text output by *appl_LCD_writeText()*.

Syntax

```
void appl_LCD_setTextCursorPos(uint16_t tcpos);
```

Parameter

```
tcpos
    New position of the text cursor;
    Range: LCD_MAX_LINES * MAX_CHAR_PRO_LINE_12 -1 (0-223)
```

Example

```
appl_LCD_setTextCursorPos(0);
```

Note: Then number of lines and characters per line is mode dependent!

appl_LCD_setTextSurface()

This function configures the typeface to be used for the text output with the functions *appl_LCD_writeText()* or *appl_LCD_showSoftkey()*.

Syntax

```
void appl_LCD_setTextSurface(appl_text_section_t section,
appl_typeface_t font);
```

Parameter

```
section
    Output section specified as value of appl_text_section_t

font
    Typeface specified as value of type appl_typeface_t
```

Example

```
appl_LCD_setTextSurface(TEXT_SECTION, FONT_NORMAL);
appl_LCD_setTextSurface(SOFTKEY_SECTION, FONT_NORMAL_BOLD);
```

appl_LCD_writeText()

This function outputs the specified text beginning at the current cursor position into the text field while taking into account the configured display mode. At the end of a line, the text is continued in the next line. The newline-character \n forces a premature line break. Usually, only an update of changed content appears on the display. This very efficient procedure restricts to fixed letter spacing and the display modes available.

Syntax

```
void appl_LCD_writeText(const char *text);
```

Parameter

```
text
    Pointer to the zero-terminated string containing the text to display.
```

Example

```
appl_LCD_writeText("Settings");
```

appl_LCD_centerText()

This function displays the specified UTF text in the centre of the text section. The output can occur in a maximum of 6 centred lines, the newline-character forces a line break.

Syntax

```
void appl_LCD_centerText(appl_typeface_t font, char *text);
```

Parameter

```
font
    Typeface of the text in accordance with type appl_typeface_t

text
    Pointer to the zero-terminated string containing the text to display.
```

Example

```
appl_LCD_centerText(FONT_LARGE,"Text");
```

appl_LCD_displayDefinedChar()

An ASCII character from the predefined font is written to the specified position. This function exists for compatibility reasons and extremely restricts the normal function of the text output.

Syntax

```
void appl_LCD_displayDefinedChar(uint16_t X, uint16_t Y, char ascii, appl_typeface_t
font, appl_status_t cu);
```

Parameter

```
x
    X-coordinate of the start of output (Range: 0 to LCD_MAX_X-1 (0-239)).

y
    Y-coordinate of the start of output (Range: 0 to LCD_MAX_Y-1 (0-319)).

ascii
    ASCII code of the character to display.

font
    Typeface of the text in accordance with type appl_typeface_t

cu
    cursor state in accordance with type appl_status_t
```

Example

```
appl_LCD_displayDefinedChar(10,10,'A',FONT_NORMAL,ON);
```

appl_LCD_displayText()

The text defined in the structure *appl_text_description_s* is directly written at the specified coordinates. The text memory of the text modes remains unchanged. This is the most flexible way of text output; the user himself is responsible for refresh actions.

Syntax

```
void appl_LCD_displayText(appl_text_description_s *td);
```

Parameter

```
td
    Pointer to the structure of the type appl_text_description_s
```

Example

```
text_description_s  text_description;

appl_LCD_clear(White);
text_description.text = "First line";
text_description.x = 5;
text_description.y = 30;
text_description.cursor = 0;
text_description.shape = SH_ORDINARY;
text_description.font = FONT_NORMAL;

appl_LCD_displayText(&text_description);
text_description.y += 15;
text_description.text = "Second line";

appl_LCD_displayText (&text_description);
```

appl_LCD_textBox()

SDK API 3

The function calculates the text box used to draw text specified with *appl_text_description_s.*

Syntax

```
void appl_LCD_textBox(appl_text_description_s *td, uint16 *pwidth, uint16_t pheight);
```

Parameter

```
td
    Pointer to the structure of the type appl_text_description_s


pwidth
    pointer to a width variable or NULL, is set to the needed width, result can be
larger than the display


pheight
    pointer to the height variable of NULL, is set to the needed height, result can be
larger than the display
```

Example

```
text_description_s  text_description;
uint16_t width;

appl_LCD_clear(White);
text_description.text = "Centered";
text_description.x = 0;
text_description.y = 30;
text_description.cursor = 0;
text_description.shape = SH_ORDINARY;
text_description.font = FONT_NORMAL;

appl_LCD_textBox(&Text_description, &widht, NULL);
text_description.x = (LCD_MAX_X - widht) / 2;
appl_LCD_displayText(&text_description);
```

appl_LCD_enableCursor()

This function turns the display of the text cursor for text modes on or off.

Syntax

```
void appl_LCD_enableCursor(appl_status_t  cue);
```

Parameter

```
cue
    Cursor status specified as value of type appl_status_t
    ON      display cursor
    OFF     display no cursor
```

Example

```
appl_LCD_enableCursor(ON);
```

appl_LCD_scrText()

SDK API Version 4: appl_LCD_scrTextExt to set style and location parameters

This function makes the specified text scroll through one line of the display according to the parameters that accompany it within the passed struct argument.

Syntax

```
void appl_LCD_scrText(appl_scr_text_s *scrt);

void appl_LCD_scrTextExt(appl_scr_text_ext_s *scrt);
```

Parameter

```
scrt
    Pointer to a scroll object.
```

Parameter

```
// The appl_scr_text_s structure is deprecated but kept for compatibility. The code
below demonstrates the compatibility:
static void appl_LCD_scrText (const appl_scr_text_s * scrt)
{
    appl_scr_text_ext_s sc = { .step_ms = scrt->step * 5, .font = scrt->font, .shape =
SH_VARIABLE, .x = 0, .y = scrt->line * 24, .width = displayMaxWidth(), .text = scrt-
>text };
    if (BOLD == scrt->size)
    {
        if (FONT_SMALL == scrt->font) { sc.font = FONT_SMALL_BOLD; }
        else if (FONT_NORMAL == scrt->font) { sc.font = FONT_NORMAL_BOLD; }
    }
    appl_LCD_scrTextExt(&sc);
}
```

Example

```
appl_scr_text_ext_s  scrt;

scrt.step_ms  = 5;                  // Shift Time Step: 5 ms
scrt.font  = FONT_LARGE;
scrt.shape = SH_VARIABLE;           // do not use monospace settings
scrt.x  = 0;
scrt.y = 20;
scrt.width = LCD_MAX_X - 2 * 20;                // Scrolling Text on display top with
left/right 20 pixel free space
scrt.text  = "A long, large text demonstrating the scrolling feature of the
display...";

appl_LCD_scrTextExt (&scrt);
```

appl_LCD_scrClear()

This function stops the output of scrolling text and clears the internal line buffer. The previously occupied display line is now available for other output.

Syntax

```
void appl_LCD_scrClear(void);
```

Example

```
appl_LCD_scrClear();
```

appl_textFontWidth()

SDK API 3

This function returns the size of a font in display pixel.

Syntax

```
uint8_t appl_textFontWidth(appl_typeface_t font);
```

Parameter

```
font
    one of the fonts
```

Reply

```
uint8_t
    font width in pixel for monospace usage
```

Example

```
uint8 width = appl_textFontWidth(FONT_NORMAL);
```

appl_textFontHeight()

SDK API 3

This function returns the size of a font in display pixel.

Syntax

```
uint8_t appl_textFontHeight(appl_typeface_t font);
```

Parameter

```
font
    one of the fonts
```

Reply

```
uint8_t
    font height in pixel, normally also the line height
```

Example

```
uint8 height = appl_textFontHeight(FONT_NORMAL);
```

Colours

The types and functions described here are used for the colour settings of all parts of the display output.

Encoding of Colours

Colours can be defined with custom values in a colour depth of up to 16 Bit in accordance with the following coding:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	r	r	r	r	r	g	g	g	g	g	g	b	b	b	b	b
	Red					Green						Blue				

Pre-defined constants are available for 10 common standard colours:

Syntax

```
WHITE/White         0xFFFF      // White
BLACK/Black         0x0000      // Black
GREY/Grey           0xF7DE      // light Grey
BLUE/Blue           0x001F      // Blue
BLUE2/Blue2         0x051F      // Light Blue
RED/Red             0xF800      // Red
MAGENTA/Magenta     0xF81F      // Magenta
GREEN/Green         0x07E0      // Green
CYAN/Cyan           0x7FFF      // Cyan
YELLOW/Yellow       0xFFE0      // Yellow
```

Defined Types

The following types have been defined for the use by color functions:

appl_color_section_t	typedef enum

appl_color_section_t

The values of this enum identify all the defined display sections that can be assigned a separate colour. The selection of a colour is available for

- text and its background,
- icons and their background,
- soft key text and its background, and
- graphics output (foreground)

Colours are encoded as 16-bit values as specified by Encoding of Colours.

Syntax

```
typedef enum
{   TEXT_COLOR,
    TEXT_BACKGROUND_COLOR,
    ICON_COLOR,
    ICON_BACKGROUND_COLOR,
    SOFTKEY_COLOR,
    SOFTKEY_BACKGROUND_COLOR,
    GRAPHIC_COLOR
}   appl_color_section_t;
```

Parameters

```
TEXT_COLOR
    Colour used for text output.

TEXT_BACKGROUND_COLOR
    Colour used for the background of text output.

ICON_COLOR
    Colour used for the display of icons.

ICON_BACKGROUND_COLOR
    Colour used for the background of icons.

SOFTKEY_COLOR
    Colour used for the display of soft key labels.

SOFTKEY_BACKGROUND_COLOR
    Colour used for the background of soft key labels.

GRAPHIC_COLOR
    Colour used for graphics output.
```

Functions

The following COLOR functions are provided:

appl_LCD_setTextColor()
appl_LCD_setBackColor()
appl_LCD_Color()

appl_LCD_setTextColor()

Configures the colour for pixel-based, direct text output. The colour settings are used immediately. Subsequently output text will be displayed in the configured colour.

Syntax

```
void appl_LCD_setTextColor(uint16_t color);
```

Parameter

```
color
    16-bit value as specified by Encoding of Colours.
```

Example

```
appl_LCD_setTextColor(Black);
```

appl_LCD_setBackColor()

This function configures the colour painted in the background of pixel-based text output. Subsequently output text will be displayed on top of a background painted in the configured colour.

Syntax

```
Syntax:    void appl_LCD_setBackColor(uint16_t color);
```

Parameter

```
color
    16-bit value as specified by Encoding of Colours.
```

Example

```
appl_LCD_setBackColor(White);
```

appl_LCD_Color()

This function re-sets colours for the specified display area and the area in question, with exception of GRAPHIC_COLOR, is updated.

Syntax

```
void appl_LCD_Color(appl_color_section_t section, uint16_t color);
```

Parameter

```
section
    Display section specified as value of type appl_color_section_t.

color
    16-bit value as specified by Encoding of Colours.
```

Example

```
appl_LCD_Color(ICON_COLOR, Blue);
```

LED

This section describes how to control the lighting of the display and the keypad.

Defined Types

The following type names have been predefined for the LED section:

appl_ledCommand_t	typedef enum

appl_ledCommand_t

This enum identifies the commands available for the lighting control.

Syntax

```
typedef enum
{   LED_CMD_OFF,
    LED_CMD_ON,
    LED_CMD_DARK,
    LED_CMD_LIGHT
}   appl_ledCommand_t;
```

Parameter

```
LED_CMD_OFF
    Turn lighting OFF and keep the old state in mind.

LED_CMD_ON
    Turn lighting ON with last settings in mind.

LED_CMD_DARK
    Turn lighting ON with reduced brightness of LCD.

LED_CMD_LIGHT
    Turn lighting ON with full brightness of LCD.
```

Functions

The following LED functions are provided:

appl_LED_control()

appl_LED_control()

This function switches the LED lighting according to the specified command.

Syntax

```
void appl_LED_control(appl_ledCommand_t cmd);
```

Parameter

```
cmd
    Command specified as value of type appl_ledCommand_t
```

Example

```
appl_LED_control(LED_CMD_OFF);
```

Graphics

This section describes the provided means for graphics output.

Defined Types

The following types have been defined for the use by graphics functions:

appl_direction_t	typedef enum

appl_direction_t

Values of this enum specify the direction of lines drawn in parallel to one of the axes.

Syntax

```
typedef enum
{   DIR_HORIZONTAL,
    DIR_VERTICAL
}   appl_direction_t;
```

Parameter

```
DIR_HORIZONTAL
    Horizontal (X) direction, from left to right.

DIR_VERTICAL
    Vertical (Y) direction, from top to bottom.
```

Functions

The following GRAPHICS functions are provided:

appl_LCD_drawRect()
appl_LCD_DrawCircle()
appl_LCD_DrawLine()
appl_LCD_writePixelPrepare()
appl_LCD_writePixel()
appl_LCD_setPixelPosition()

appl_LCD_drawRect()

This function draws a rectangle (filled box). It is specified by the coordinates of its top left corner.

Syntax

```
void appl_LCD_drawRect(uint16_t x, uint16_t y, uint16_t height, uint16_t width);
```

Parameter

```
x
    X-coordinate of the top left corner point; Range: 0 to LCD_MAX_X-1

y
    Y-coordinate of the top left corner point; Range: 0 to LCD_MAX_Y-1

height
    Height of the rectangle in pixels; Range: 1 to LCD_MAX_Y

width
    Width of the rectangle in pixels; Range: 1 to LCD_MAX_X
```

Example

```
appl_LCD_drawRect(10, 10, 30, 20);
```

appl_LCD_DrawCircle()

This function draws a circle (just the rand, not filled) specified by its center and its radius.

Syntax

```
void appl_LCD_DrawCircle(uint16_t x, uint16_t y, uint16_t radius);
```

Parameter

```
X
    X-coordinate of the center point; Range: 0 to LCD_MAX_X-1

Y
    Y-coordinate of the center point; Range: 0 to LCD_MAX_Y-1

radius
    Radius of the circle in pixels; Range: 1 upward
```

Example

```
appl_LCD_DrawCircle(50, 50, 20);
```

appl_LCD_DrawLine()

This function draws a line from the specified starting point and of the specified length. The coordinates refer to the start point on the left or on top.

Syntax

```
void appl_LCD_DrawLine(uint16_t x, uint16_t y, uint16_t length, appl_direction_t
direction);
```

Parameter

```
x
    X-coordinate of the starting point; Range: 0 to LCD_MAX_X-1

y
    Y-coordinate of the starting point; Range: 0 to LCD_MAX_Y-1

length
    Length of the line in pixels; Range: 1 to LCD_MAX_Y

direction
    The direction of the line to be drawn specified as value of type appl_direction_t
```

Example

```
appl_LCD_DrawLine(10, 10, 30, DIR_HORIZONTAL);
```

Pixel based graphic functions

Drawing a graphics requires the definition of an area (rectangle), a start command for the memory transfer, transferring all pixels (ideal with a DMA, here with a function call per pixel), and normally an end command of the transfer (can be omitted because other display command interrupts the transfer).

Until SDK API 4 the area was restricted to a 1x1 rectangles and the transfer of exactly on pixel (appl_LCD_setPixelPosition, appl_LCD_writePixelPrepare, appl_LCD_writePixel). That is rather inefficient. With SDL

API 4 the area function has been added. This allows the implementation of non DMA based graphics ins sufficient fast manor (appl_LCD_setPixelArea, appl_LCD_writePixel, appl_LCD_writePixel, ...)

appl_LCD_setPixelArea()

This function prepares the display for the output of pixel data. Technically, the display is put into data mode and then interprets the next data as colours.

Syntax

```
void LCD_setPixelArea(uint16_t Xpos, uint16_t Ypos, uint16_t width, uint16_t height);
```

Parameter

```
Xpos, Ypos
    left upper corner of the pixel area

width, height
    dimension of the pixel area (width * height pixel data are accepted now)
```

Example

```
LCD_setPixelArea(10, 10, 30, 30);
for(unsigned i = 0; i < 30 * 30; i++) { appl_LCD_writePixel(GREEN); } // fills a green
rectangle 30x30 pixel
```

appl_LCD_writePixelPrepare()

This function prepares the display for the output of pixel data. Technically, the display is put into data mode and then interprets the next data as colours.

Syntax

```
void appl_LCD_writePixelPrepare(void);
```

Example

```
appl_LCD_writePixelPrepare();
```

appl_LCD_writePixel()

This function outputs a single pixel in the specified colour. This only works when the display is in data mode. It is recommended to avoid writing many single pixels. The function attends the compatibility flag for the colour.

Syntax

```
void appl_LCD_writePixel(uint16_t color);
```

Parameter

```
color
    16-bit value as specified by Encoding of Colours.
```

Example

```
appl_LCD_writePixel(Green);
```

appl_LCD_setPixelPosition()

This function places the pixel cursor at the specified position so that the next output of a pixel will occur there. Technically, a data window, with the length of one pixel and the width of one pixel is created. This is an extremely inefficient process!

Syntax

```
void appl_LCD_setPixelPosition(uint16_t x, uint16_t y);
```

Parameter

```
x
    New x-coordinate of the pixel cursor; Range: 0 to LCD_MAX_X-1

y
    New y-coordinate of the pixel cursor; Range: 0 to LCD_MAX_Y-1
```

Example

```
appl_LCD_setPixelPosition(10, 10);
```

Soft Keys

This section describes the control of the text fields used for soft keys.

Defined Types

The following types have been defined for the use by soft key functions:

appl_sk_side_t	typedef enum
appl_sk_property_t	typedef enum

appl_sk_side_t

The values of this enum identify one of the two text fields available for soft key labels.

Syntax

```
typedef enum
{
    SK_NONE,
    SK_LEFT,
    SK_RIGHT,
    SK_BOTH
}   appl_sk_side_t;
```

Parameter

```
SK_NONE
    No soft key selected
SK_LEFT
    Label of left soft key.

SK_RIGHT
    Label of right soft key.


SK_BOTH
    Both soft keys selected.
```

appl_sk_property_t

The values of this enum distinguish the static and the blinking display of the label text.

Syntax

```
typedef enum
{
    SK_STATIC,
    SK_BLINKING
}   appl_sk_property_t;
```

Parameter

```
SK_STATIC
    Static label text.

SK_BLINKING
    Blinking label text.
```

Functions

The following SOFTKEY functions are provided:

appl_LCD_showSoftkey()
appl_LCD_setSoftkeyProperty()

appl_LCD_showSoftkey()

This function updates the text of the specified soft key label.

Syntax

```
void appl_LCD_showSoftkey(appl_sk_side_t side, char *text);
```

Parameter

```
side
    Selects the soft key label to update as value of type appl_sk_side_t.

text
    Points to the zero-terminated string containing the new soft key label.
    Maximum length: depends on used font and characters (variable font setting)
```

Example

```
appl_LCD_showSoftkey(SK_LEFT, "OK");
```

appl_LCD_setSoftkeyProperty()

This function updates how the specified soft key label is displayed.

Syntax

```
void appl_LCD_setSoftkeyProperty(appl_sk_side_t side, appl_sk_property_t pr);
```

Parameter

```
side
    Selects the soft key label to update as value of type appl_sk_side_t

pr
    Specifies how to display the label as value of type appl_sk_property_t
```

Example

```
appl_LCD_setSoftkeyProperty(SK_RIGHT, SK_STATIC);
```

Icons

The upper part of the display of the HA40 can be used for the display of pre-defined icons. This section describes the handling of such icons.

Defined Types

The following types have been defined for the use by icon functions:

appl_icon_t	typedef enum
appl_rocker_mode_t	typedef enum

appl_icon_t

The values of this enum identify several system-defined icons:

- Field strength (signal strength)
- Roaming
- Numeric digit
- SMS
- Audio mode
- Volume and
- Mute

Syntax

```
typedef enum
{   ICON_SIGNALSTRENGTH,
    ICON_ROAMING,
    ICON_MCALLS,
    ICON_SMS,
    ICON_VOLUME_HF,
    ICON_VOLUME_PR,
    ICON_MICRO_MUTE,
    ICON_AUDIO_MODE
}   appl_icon_t;
```

Parameter

```
ICON_SIGNALSTRENGTH
    Icon for the strength of the GSM signal. Supports a value from 0-6 for raising
bars.

ICON_ROAMING
    Roaming icon. A switch, value 1 for shown.

ICON_MCALLS
    Single numeric digit, normally missed calls until 9 are displayed.

ICON_SMS
    SMS icon (envelope symbol) 1 read SMS, 2 unread SMS, 3 read blinking, 4 unread
blinking

ICON_VOLUME_HF
    Icon for the volume of the hands-free set, volume icon with value 1-10 is
displayed.

ICON_VOLUME_PR
    Icon for the volume of the handset, volume icon with value 1-10 is displayed.

ICON_MICRO_MUTE
    Icon for the microphone mute. A switch, value 1 if shown.

ICON_AUDIO_MODE
    Icon for the audiomode, 1 handsfree, 2 private
```

The graphic representation of these icons is shown in the User Manual. A value 0 deactivates the icon.

appl_rocker_mode_t

An icon displaying arrows for the rocker navigation may be shown between the text fields of the soft key labels. The values of this enum identify the available symbols.

Syntax

```
typedef enum
{
    ROCKER_OFF,                         ///< No icon
    ROCKER_DOWN,                        ///< Arrow to down
    ROCKER_UP,                          ///< Arrow to up
    ROCKER_UP_DOWN,                     ///< Double arrow up and down
    ROCKER_LEFT,                        ///< Arrow to left
    ROCKER_RIGHT,                       ///< Arrow to right
    ROCKER_LEFT_RIGHT,                  ///< Double arrow left and right
    ROCKER_ALL,                         ///< Both double arrows
    ROCKER_UP_LEFT,                     ///< Double arrow up left
    ROCKER_UP_RIGHT,                    ///< Double arrow up right
    ROCKER_UP_LEFT_RIGHT,               ///< Triple arrow up left right
    ROCKER_DOWN_LEFT,                   ///< Double arrow down left
    ROCKER_DOWN_RIGHT,                  ///< Double arrow down right
    ROCKER_DOWN_LEFT_RIGHT,             ///< Triple arrow down left right
    ROCKER_UP_DOWN_LEFT,                ///< Triple arrow up down left
    ROCKER_UP_DOWN_RIGHT,               ///< Triple arrow up down right
    ROCKER_RETURN,                      ///< Return symbol
    ROCKER                              ///< Last icon number for iterators
}   appl_rocker_mode_t;
```

Parameter

```
ROCKER_...
    see comment within the enum definition.
```

The graphical representation of these navigation icons is shown in the User Manual.

Functions

The following ICON functions are provided:

appl_LCD_setIcon()
appl_LCD_drawRocker()

appl_LCD_setIcon()

This function displays a pre-defined icon.

Syntax

```
Syntax:    void appl_LCD_setIcon(appl_icon_t icon, uint8_t value);
```

Parameter

```
icon
    Specifies the icon to display as value of type appl_icon_t

value
    Value to be assigned to the icon. The valid range depends on the selected icon.
Value 0 deactivates the icon.
```

Example

```
appl_LCD_setIcon(ICON_ROAMING, 1);
```

appl_LCD_drawRocker()

This function updates the icon displayed for the rocker navigation.

Syntax

```
void appl_LCD_drawRocker(appl_rocker_mode_t mode);
```

Parameter

```
mode
    The rocker navigation to offer specified as a value of type appl_rocker_mode_t
```

Example

```
appl_LCD_drawRocker(ROCKER_ALL);
```

Images and Bitmaps

The handset is capable of displaying images. It can handle colour images in BMP format, compressed BMP as well as monochrome bitmaps. A logo is a special BMP image, which is provided by the system.

Defined Types

The following types have been defined for the use by image and bitmap functions:

logo_t	typedef enum
appl_bmp_description_s	typedef struct
appl_bm_description_s	typedef struct

logo_t

The values of this enum select the logo to display. The default logo is the one of pei tel. A custom logo may also be available. If no custom logo has been stored to the device flash, the pei tel logo will be displayed regardless of the selection.

Syntax

```
typedef enum
{   DEFAULT_LOGO,
    CUSTOMER_LOGO
}   logo_t;
```

Parameter

```
DEFAULT_LOGO
    Select the pei tel logo.

CUSTOMER_LOGO
    Select the custom logo.
```

appl_bmp_description_s

This struct summarizes the properties of a stored image (BMP file). As usual for BMP images, the coordinate origin is the lower left corner. But, the position of the image is given in relation to [0,0] at the left upper corner of the display.

Syntax

```
typedef struct
{   uint16_t * bmpAddress;
    uint8_t xPos;
    uint8_t yPos;
    uint8_t xDots;
    uint8_t yDots;
    bool byteOrder;
}   appl_bmp_description_s;
```

Parameter

```
bmpAddress
    Pointer to 16-bit encoded BMP data.

xPos
    X-coordinate of image origin; Range: 0 to LCD_MAX_X-1

yPos
    Y-coordinate of image origin; Range: 0 to LCD_MAX_Y-1

xDots
    Width of image in pixels; Range: 1 to LCD_MAX_X

yDots
    Height of image in pixels; Range: 1 to LCD_MAX_Y

byteOrder
    Alignement of colour values: if set to TRUE, the MSB and the LSB of the 16-bit
values will be exchanged.
```

appl_bm_description_s

This function is kept for compatibility reasons. Modern system would use UTF symbols and set respective text. The UTF bitmaps of the device can be extended as desired.

This structure describes the properties of a user-defined bitmap image. The image is encoded line by line and bit by bit within a line. The individual lines are joined without gaps to form a compact bitmap representation. The bit stream is stored as a vector of 32-bit values.

Example: Definition of a new symbol of 6 by 5 pixels.

					Coding:
					1 1 1 1 1 b
					1 0 0 0 1 b
					1 0 0 0 1 b
					1 0 0 0 1 b
					1 0 0 0 1 b
					1 1 1 1 1 b

Resulting compact bitmap representation:

1	1	1	1	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	1	1	1	1	x	x
F				C				6				3				1				8				F				C			
0xFC								0x63								0x18								0xFC							

Table 5: Encoding of a Bitmap

x = don’t care

Bitmap: u32 square[1] = {0xFC6318FC};

The encoding of the image starts at the upper left corner. A set bit [1] represents a pixel of the text colour, a cleared bit [0] represents a pixel to be drawn in the background colour. The bitmap data must be padded to the next byte boundary. The values of possibly added bits are irrelevant.

Syntax

```
typedef struct
{   u32 *bitmap;
    uint8_t width;
    uint8_t height;
    uint16_t x;
    uint16_t y;
    uint16_t pixel_color;
    uint16_t back_color;
    appl_shapes_t shape;
}   appl_bm_description_s;
```

Parameter

```
bitmap
    Pointer to the compact bitmap encoding of the image.

width
    Width of the image in pixels; Range: 1 to LCD_MAX_X

height
    Height of the image in pixels; Range: 1 to LCD_MAX_Y

x
    X-coordinate of image origin; Range: 0 to LCD_MAX_X-1

y
    Y-coordinate of image origin; Range: 0 to LCD_MAX_Y-1

pixel_color
    Colour of active pixels (foreground)

back_color
    Background colour for non-active pixels

shape
    Display mode specified as value of type appl_shapes_t restructed to SH_NORMAL,
SH_INVERS, SH_TRANSPARENT
```

Functions

The following IMAGE and BITMAP functions are provided:

appl_LCD_writeBMP()
appl_LCD_writeBitmap()
appl_LCD_writeLogo()

appl_LCD_writeBMP()

This function draws the BMP image (without BMP header) as specified by the passed appl_bmp_description_s parameter. The BMP data has to be passed so that it results in a "little endian" format. As the storage is done mostly in "big endian", *byteOrder* must be set to TRUE. The colour coding RGB565 is mandatory and incompatible with the HA57 SDK (RGB555) meaning the images need to be recreated with adjusted colour space and resolution.

Syntax

```
Syntax:    void appl_LCD_writeBMP(appl_bmp_description_s *bmp);
```

Parameter

bmd
    Pointer to an appl_bmp_description_s struct.

Example

```
appl_bmp_description_s bmp_description;

bmp_description.bmpAddress = &pct;
bmp_description.xPos = 0;
bmp_description.yPos = 0;
bmp_description.xDots = LCD_MAX_X;
bmp_description.yDots = LCD_MAX_Y;
bmp_description.byteOrder = TRUE;

appl_LCD_writeBMP(&bmp_description);
```

API VERSION 4

The firmware of HA40 supports a pei tel specific decompression, and optimizes for a small memory footprint and non-photographic BMP's. Logo files (e.g. the pei tel logo) can be reduced up to 1:100. Contact the pei tel support with a BMP picture drawable using appl_LCD_writeBMP to get back a compressed image within a C file. The compressed file includes the BMP header and uses the height, width and some other parameters to display the image. In case of header or decompress errors, false is returned (a correct usage should always return true).

Syntax

```
Syntax:    bool appl_LCD_drawCompressedBMP(uint16_t x, uint16_t y, const void* bmp,
unsigned len);
```

Example

```
static const uint8_t peitel[2923] = {
    0x20, 0xA8, 0x87, 0x2D, 0x6B, 0xFF, 0x9E, 0x6D, 0x6B, 0xF6, 0x06, 0xA0, 0xAB, 0x9E,
 0xFF, 0x92,
    0x94, 0xB1, 0xFF, 0xEF, 0xC2, 0xFF, 0xFC, 0xAB, 0xB1, 0xFC, 0xF6, 0x87, 0xA8, 0x9C,
 0xF3, 0xFF,
    ...
    0x15, 0x13, 0xF4, 0x83, 0x06, 0x5B, 0xFF, 0x89, 0xB8, 0x9A, 0xBA, 0x9F, 0xA4, 0x95,
 0x7F, 0xF1,
    0x37, 0x2E, 0xDD, 0xD3, 0xF4, 0x6E, 0xC5, 0xCB, 0xFF, 0x89, 0xC8, 0xF4, 0xEE, 0x36,
 0xAE, 0x99,
    0x0A, 0x80, 0x4A, 0xC6, 0x61, 0xFF, 0xE1, 0xD8, 0xBC, 0x00, 0xC0,
};

if (!appl_LCD_drawCompressedBMP(0, 0, peitel, sizeof(peitel)) { /* ops */ }
```

appl_LCD_writeBitmap()

This function draws a bitmap as specified by the passed *appl_bm_description_s* parameter.

Syntax

```
void appl_LCD_writeBitmap(appl_bm_description_s *bm);
```

Parameter

bm
    Pointer to the bitmap see section Encoding of Colours

Example

```
appl_bm_description_s  bm_description;
uint8_t square[4] = {0xFC, 0x63, 0x18, 0xFC};

bm_description.bitmap = square;
bm_description.width = 5;
bm_description.height = 6;
bm_description.x = 20;
bm_description.y = 30;
bm_description.shape = SH_ORDINARY;

appl_LCD_writeBitmap(&bm_description);
```

appl_LCD_writeLogo()

This function displays the specified logo.

Syntax

```
void appl_LCD_writeLogo(logo_t logo);
```

Parameter

```
logo
    The logo to display specified as value of type logo_t
```

Example

```
appl_LCD_writeLogo(CUSTOMER_LOGO);
```

System Information

This section describes how to retrieve system information and relevant hardware addresses. The returned values reflect the equipment of the hosting device and may differ on non-standard models.

Defined Types

The following types have been defined for the use by system information functions:

sys_parameter_t	typedef enum

sys_parameter_t

This struct contains the basic addresses and capacity figures of a device.

Syntax

```
typedef enum
{   RAM_BASE_ADDRESS,
    FLASH_BASE_ADDRESS,
    RAM_SIZE,
    FLASH_SIZE,
    FLASH_SECTOR_SIZE,
    FLASH_BLOCK_SIZE
}   sys_parameter_t;
```

Parameter

```
RAM_BASE_ADDRESS
    Base address of the random-access memory (RAM) for applications.

FLASH_BASE_ADDRESS
    Base address of the program flash memory for applications.

RAM_SIZE
    Size of the available RAM for applications.

FLASH_SIZE
    Size of a separate flash memory (not included) ->0

FLASH_SECTOR_SIZE
    Sector size of the flash memory -> 0

FLASH_BLOCK_SIZE
    Block size of the flash memory -> 0
```

Functions

The following SYSTEM INFORMATION functions are provided:

appl_System_getParameter()

appl_System_getParameter()

This function retrieves the basic system parameters.

Syntax

```
u32 appl_System_getParameter(sys_parameter_t par);
```

Parameter

```
par
    Pointer to the sys_parameter_t struct, which is to be filled with the system
parameters.
```

Example

```
sys_parameter_t sys_parameter;
appl_System_getParameter(&sys_parameter);
```

Timer

This section describes how timers can be utilized in the application design.

Defined Types

The following types have been defined for the use by timer functions:

appl_system_timer_t	typedef enum

appl_system_timer_t

Six (6) system timers are provided for the use by an application. The KEY_TIMER should be used to control the repeat rate of the keypad; the LOGO_TIMER should control the display of the logo; and the LIGHT_TIMER should control the lighting. The three USER_TIMERx are available for free usage.

Timers on SDK API are one-shot timers and have to be restarted for cyclic timeouts.

Syntax

```
typedef enum
{
    KEY_TIMER,
    LIGHT_TIMER,
    LOGO_TIMER,
    USER_TIMER1,
    USER_TIMER2,
    USER_TIMER3
}   appl_system_timer_t;
```

Parameter

```
KEY_TIMER
    Timer for controlling the key repeat rate. 1 Tick 100ms

LIGHT_TIMER
    Timer for controlling the lighting. 1 Tick 100ms

LOGO_TIMER
    Timer for controlling the display of the logo. 1 Tick 100ms

USER_TIMER1
    Freely available Timer1 1 Tick 100ms

USER_TIMER2
    Freely available Timer2 1 Tick 100ms

USER_TIMER3
    Freely available Timer3 1 Tick 10ms
```

Note: Logo-, key- and light-timers are not started automatically for SDK program applications. However for protocol extensions these timers are maintained by the protocol implementation itself.

Functions

The following TIMER functions are provided:

appl_Timer_delay()
appl_Timer_start()
appl_Timer_stop()
appl_Timer_is()
API version 4: appl_Timer_start_ms()

appl_Timer_delay()

This function suspends the user application for the specified number of system ticks. The duration of a system tick is 10 ms. This function especially serves the Watchdog for freeze detection.

Syntax

```
void appl_Timer_delay(unsigned int tick);
```

Parameter

```
tick
    Number of system ticks.
```

Example

```
appl_Timer_delay(100);

// Wait for 1 s (= 100 * 10 ms)
```

appl_Timer_start()

This function starts the specified timer with the specified target running time. The running time is specified in timer ticks. The duration of a timer tick is 100 ms for the timers *KEY_TIMER*, *LIGHT_TIMER*, *LOGO_TIMER*, *USER_TIMER1*, and *USER_TIMER2*. It is 10 ms for the timer *USER_TIMER3*.

Syntax

```
void appl_Timer_start(unsigned int tick, appl_system_timer_t tm);

void appl_timer_start_ms(unsigned int ms, appl_system_timer tm);
```

Parameter

```
tick
    Number of timer ticks (timer depended)

ms
    Timer duration in milliseconds (since API version 4). The timer resolution in HA40
is always 1 ms.

tm
    Identifies the timer to use by a value of type appl_system_timer_t.
```

Example

```
appl_Timer_start(30, LOGO_TIMER);
appl_Timer_start_ms(250, LIGHT_TIMER)

// Time out in 3 s (= 30 * 100 ms) or 250 ms
```

appl_Timer_stop()

This function stops the specified timer. The stopped timer will not generate a timeout event. Already expired timers do not need to be stopped explicitly, which means, timers are never cyclic.

Syntax

```
void appl_Timer_stop(appl_system_timer_t tm);
```

Parameter

```
tm
Identifies the timer to stop by a value of type appl_system_timer_t
```

Example

```
appl_Timer_stop(USER_TIMER1);
```

appl_Timer_is()

This function determines whether the specified (system) timer is currently running (TRUE) or not (FALSE). This call can be very quick, it is possible that the timeout event is already queued for delivery to application and the timer is not running anymore.

Syntax

```
bool appl_Timer_is(appl_system_timer_t tm);
```

Parameter

```
tm
    Identifies the timer to query by a value of type appl_system_timer_t
```

Reply

```
bool
    Timer activity: TRUE - running or FALSE - not running.
```

Example

```
if (appl_Timer_is(USER_TIMER1))
{    appl_Timer_stop(USER_TIMER1);
}
```

UART

UART connection serves as the central control interface to the handset. The initialization and utilization of this interface are described in this section.

The standard initialization is 115200/8/N/1 and always active together with the start event. Reading and writing is buffered (range is 1-2 Kbyte, the value might be changed for firmware updates).

Defined Types

The following types have been defined for the use by UART functions:

UART_Parity_t	typedef enum
UART_StopBits_t	typedef enum
UART_WordLength_t	typedef enum

UART_Parity_t

The values of this enum identify the parity setting of the UART transmission.

Syntax

```
typedef enum
{   UART_NO_PARITY,
    UART_EVEN_PARITY,
    UART_ODD_PARAITY
}   UART_Parity_t;
```

Parameter

```
UART_NO_PARITY
    No parity checking.

UART_EVEN_PARITY
    Even parity.

UART_ODD_PARITY
    Odd parity.
```

UART_StopBits_t

The values of this enum identify the duration of the stop bit.

Syntax

```
typedef enum
{
    UART_0_5_STOPBIT,
    UART_1_STOPBIT,
    UART_1_5_STOPBIT,
    UART_2_STOPBIT
}   UART_StopBits_t;
```

Parameter

```
UART_0_5_STOPBIT
    Duration of the stop bit ½ bit time, no hardware support for HA40, set to 1 if
 used

UART_1_STOPBIT
    Duration of the stop bit 1 bit time.

UART_1_5_STOPBIT
    Duration of the stop bit 1.5 bit times, no hardware support for HA40, set to 1 if
 used

UART_2_STOPBIT
    Duration of the stop bit 2 bit times.
```

UART_WordLength_t

This enum identifies the number of bits used for the transfer of one character (data byte).

Syntax

```
typedef enum
{
    UART_WORDLENGTH_8B,
    UART_WORDLENGTH_9B
}   UART_WordLength_t;
```

Parameter

```
UART_WORDLENGTH_8B
    A transmitted character consists of 8 bits.

UART_WORLDLENGTH_9B
    A transmitted character consists of 9 bits.
```

Functions

The following UART functions are provided:

appl_UART_sendString()
appl_UART_write()
appl_UART_clear()
appl_UART_bytesToRead()
appl_UART_init()
appl_UART_setByteState()
appl_UART_getByte()
appl_UART_read()

appl_UART_sendString()

This function transmits the specified zero-terminated string across the UART. This function is blocked until string output, when the output buffer is full.

Syntax

```
Syntax:    void appl_UART_sendString(const char *strData);
```

Parameter

```
strData
    Pointer to the zero-terminated string to be transmitted.
```

Example

```
appl_UART_sendString("Init");
```

appl_UART_write()

This function transmits *n* bytes across the UART. This function serves the transmit buffer. The transmission happens asynchronously, sometimes much later, and with full transmit buffer, the number of stored bytes might be smaller than necessary.

Syntax

```
int appl_UART_write(const void *data, int n);
```

Parameter

```
data
    Pointer to a vector of bytes.

n
    Number of bytes to be transmitted.
```

Reply

```
int
    The actual number of transferred bytes, or a negative number for errors.
```

Example

```
uint8_t buffer[5];
buffer[0] = 'O';
buffer[1] = 'K';
appl_UART_write(buffer, 2);
```

appl_UART_clear()

This function clears the receive buffer of the UART, it is recommended for the case of receive errors.

Syntax

```
Syntax:    void appl_UART_clear(void);
```

Example

```
appl_UART_clear();
```

appl_UART_bytesToRead()

This function retrieves the number of unread bytes in the receive buffer of the UART.

Syntax

```
uint16_t appl_UART_bytesToRead(void);
```

Reply

```
uint16_t
    Number of unread bytes in the buffer.
```

Example

```
uint16_t n;
n = appl_bytesToRead();
```

appl_UART_init()

This function initializes the UART using the specified parameters.

Syntax

```
void appl_UART_init(u32 br, UART_Parity_t par, UART_StopBits_t  stb, UART_WordLength_t
wl);
```

Parameter

```
br
    Baud rate.

par
    Parity specified as value of type UART_Parity_t.

stb
    Stop bits specified as value of type UART_StopBits_t.

wl
    Bit count of a character specified as value of type UART_WORDLength_t.
```

Example

```
appl_UART_init(115200, UART_NO_PARITY, UART_1_STOPBIT, UART_WORDLENGTH_8B);
```

appl_UART_getByte()

This function retrieves the next unread byte from the receive buffer. As the reception of data will generate an event, polling loops are not needed! This procedure is inefficient and should be avoided for larger amounts of data. Please take into account, that binary data can't be distinguished from error codes.

Syntax

```
Syntax:    uint8_t appl_UART_getByte(void);
```

Reply

```
uint8_t
    Next byte retrieved from the receive buffer.
```

Example

```
uint8_t c;
    c = appl_UART_getByte();
```

appl_UART_read()

This function retrieves the specified number of bytes (nbr) of the receive buffer. If successful, the return value equals the parameter "nbr". If less bytes are waiting in the buffer, the return value equals the actual amount of read bytes. It is not recommended to use this function for protocol extensions!

Syntax

```
int appl_UART_read(uint8_t * buffer, uint16_t nbr);
```

Parameter

```
buffer
    Pointer to the buffer.

nbr
    Number of bytes to be read.
```

Reply

```
int
    Value >= 0  Number of bytes read
    Value = -1  UART synchronization error
    Value = -2  Overflow receive buffer
    Value = -3  UART format error
```

Example

```
uint8_t buffer[10];
int retVal;

retVal = appl_UART_read( buffer, 10);
if ( retVal < 0 )
    {    // Error handling    }
```

appl_UART_getBuffer()

The function retrieves the actual protocol command (one line) with a maximum of specified bytes (nbr) from the decoder buffer. If successful, the length of the protocol line is returned. The decoder buffer is occupied when a

command or text was received and the reception was reported by a respective event. **This function can be respectively used for protocol extensions only.**

Syntax

```
int appl_UART_getBuffer(uint8_t * buffer, uint16_t nbr);
```

Parameter

```
buffer
    Pointer to the buffer.

nbr
    Number of bytes to be read.
```

Reply

```
int
    Value >= 0      Number of bytes read
```

Example

```
uint8_t buffer[10];
int lng;
lng = appl_UART_getBuffer( buffer, 10);
```

appl_UART_useCommand()

The emulation extensions for HA20x or HA400 can feed commands to the protocol decoder. These fed commands are interpreted as if they were received by the UART. **This function can be respectively used for protocol extensions only.**

Syntax

```
void appl_UART_useCommand(uint8_t * command );
```

Parameter

```
command
    Pointer to a zero-terminated command string.
```

Example

```
#define ESC  0x1B
uint8_t buffer[10];
sprintf(buffer,"%cIN: %u\r",ESC,5);
appl_UART_useCommand( buffer );
```

Keypad

The main user input interface is the keypad with 21 individual keys. This section describes the different input options available for this keypad. From the system's perspective, the keys are numbered in a straight sequence. The layout of the keypad is shown in the User Manual. The API 4 interface file also provides C defines for all keys. The define KEY_LAST is actually identical with KEY_EMERGENCY, this can change however with other hardware.

Pressing PTT and HOOK notification are not handled as keys, there are separate events and functions.

Key Number	Key Label	C define
0	- no key pressed -	KEY_NONE
1	0	KEY_0
2	1	KEY_1
3	2	KEY_2
4	3	KEY_3
5	4	KEY_4
6	5	KEY_5
7	6	KEY_6
8	7	KEY_7
9	8	KEY_8

Key Number	Key Label	C define
10	9	KEY_9
11	*	KEY_STAR
12	#	KEY_HASH
13	Right Soft Key	KEY_RSK
14	Left Soft Key	KEY_LSK
15	Right Function Key (ON/OFF)	KEY_RED
16	Left Function Key	KEY_GREEN
17	Arrow Up	KEY_UP
18	Arrow Down	KEY_DOWN
19	Arrow Right	KEY_RIGHT
20	Arrow Left	KEY_LEFT
21	Key emergency	KEY_EMERGENCY

Table 6: Key Codes

Functions

The following KEYPAD functions are provided:

appl_Key_getCode()
appl_Key_getHook()
app_Key_setBrightness()
app_Key_getBrightness()

appl_Key_getCode()

This function retrieves

- the key code for the currently pressed key if the event EV_KEY_INPUT or EV_KEY_TIMEOUT is set,
- new since SDK API 4: the key code for the currently released key if the event EV_KEY_RELEASED is set

- before SDK API 4: 0 if the event EV_KEY_RELEASED is set,

New since SDK API 4 is, that all pressed/released keys are reported with a separate callback. The key state (result of this function) remains active until the next key notification. Before SDK API 4 pressing multiple keys returned a key error.

Syntax

```
uint16_t appl_Key_getCode(void);
```

Reply

```
uint16_t
    Key code of the pressed key according to Table 6: Key Codes.
```

Example

```
uint16_t keyNumber;
keyNumber = appl_Key_getCode();
```

appl_Key_getHook()

This function retrieves the status of the hook switch. An ON status will be returned if the handset is mounted to the cradle, an OFF status will be returned otherwise.

Syntax

```
appl_status_t appl_Key_getHook(void);
```

Reply

```
appl_status_t
    Status of the hook switch specified as value of type appl_status_t
```

Example

```
appl_status_t hookStatus;
hookStatus = appl_Key_getHook();
```

The result of this query is independent from the settings of the hardware signals.

appl_Key_setBrightness()

The brightness of the key pad illumination is set: There are 100 levels (percent) of brightness available. The values range from 0 to KEYPAD_MAX_BRIGHTNESS. The value 0 switches the illumination off.

Syntax

```
void appl_Key_setBrightness(uint8_t brightness);
```

Parameter

```
brightness
    Brightness 0 ... 100, values above 100 are set to 100
```

Example

```
appl_Key_setBrightness ( 5 );
```

appl_Key_getBrightness()

The brightness of the keypad illumination is queried.

Syntax

```
uint8_t appl_Key_getBrightness(void);
```

Reply

```
uint8_t
    Brightness 0 ... 100
```

Example

```
brightness = appl_Key_getBrightness ();
```

Object Flash Memory

This section describes the non-volatile flash memory available for the persistent storage of parameters and configuration data (serial SPI flash). These data objects are managed through indexes ("*objIdent*"). The number of required data objects must be initialized by the application through *appl_Object_subscribe*. Each data object is uniquely identified by its index from the range 0 to object_count-1. The maximum supported object count is 256. Each object is placed onto a different page in order to avoid side effects between objects on update/delete actions.

The object flash memory is not suited for frequently changing or large data objects (100.000 flash cycles minimum, data retention: 20 years, 255 bytes maximum size per object). If a flash object change is called the result is undefined for that object only in case of interruption (e.g. a power cycle).

Note: Flash objects are not deleted by design with a factory reset from system menu!

Defined Types

The following types have been defined for the use by OBJECT-FLASH functions:

object_status_t	typedef enum

object_status_t

The values of this enum identify the return status of an operation on the object flash memory.

Syntax

```
typedef enum
{
    OBJECT_OK,
    OBJECT_EMPTY,
    OBJECT_ERROR,
    OBJECT_LENGTH
}   object_status_t;
```

Parameter

```
OBJECT_OK
    The data object is available or action was successful.

OBJECT_EMPTY
    The object is not used yet and can be written to.

OBJECT_ERROR
    An error occurred during the creation of the object.

OBJECT_LENGTH
    Bad length parameter used.
```

Functions

The following OBJECT-FLASH functions are provided:

appl_Object_subscribe()
appl_Object_read()
appl_Object_write()
appl_Object_is()

appl_Object_subscribe()

This function initializes the object flash and configures the number of required data objects. This function must only be called once during the initialization of the user application.

Syntax

```
bool appl_Object_subscribe(uint8_t numberObjects);
```

Parameter

```
numberObjects
    Number of required data objects.
```

Reply

```
bool
    TRUE is returned upon a successful initialization, FALSE otherwise.
```

Example

```
appl_Object_subscribe(5);
```

appl_Object_read()

This function reads the specified data object and copies its content to the specified data buffer. The size of the data buffer must match the size of the data object. The object identifier *objIdent* must be strictly smaller than the configured number of objects.

Syntax

```
object_status_t appl_Object_read(uint8_t objIdent, void *data, uint8_t length);
```

Parameter

```
objIdent
    Identifies the object to read.

data
    Pointer to data buffer receiving the copy of the object contents.

length
    Count of bytes to copy from the data object to the data buffer.
```

Reply

```
object_status_t
    Success status as value of type object_status_t
```

Example

```
appl_hw_description_s   hwd;
Object_status_t        os;

os = appl_Object_read(0, (uint8_t*)&hwd, sizeof(appl_hw_description_s));
```

appl_Object_write()

The content of the provided data buffer is copied into the specified object. The previous contents of the data object are silently overwritten. This is a synchronous write through to the SPI flash.

Syntax

```
object_status_t appl_Object_write(uint8_t objIdent, const void *data, uint8_t length);
```

Parameter

```
objIdent
    Identifies the object to write to.

data
    Pointer to the data buffer providing the data contents.

length
    Count of bytes to copy from the data buffer into the data object, use 0 to delete
the object
```

Reply

```
object_status_t
    Success status as value of type object_status_t
```

Example

```
appl_hw_description_s   hwd;
Object_status_t        os;

os = appl_Object_write(0, (uint8_t*)&hwd, sizeof(appl_hw_description_s));
```

appl_Object_is()

This function queries the status of a flash memory object.

Syntax

```
object_status_t appl_Object_is(uint16_t objIdent, uint8_t length);
```

Parameter

```
objIdent
    Identifies the object to query.

length
    Expected size of the specified data object or memory area in bytes.
```

Reply

```
object_status_t
    Success status as value of type object_status_t
```

Example

```
char buffer[15];

appl_Object_subscribe(6);
if  (appl_Object_is(3, 15) == OBJECT_EMPTY)
{
    appl_Object_write(3, "ABCDEFGHIJKLMNO", 15);
} else {
    appl_Object_read(3, buffer, 15);
}
```

Hardware

This section describes the functions provided for the control of the device hardware (speaker, microphone, audio amplifier) and the hook and PTT signals. A detailed block diagram is provided in the User Manual.

Defined Types

The following types have been defined for the use by HARDWARE functions:

appl_status_t	typedef enum
appl_hw_description_s	typedef struct

appl_status_t

The enum *appl_status_t* is defined for the hardware states ON and OFF and can be used as a parameter in function and other structures.

Syntax

```
typedef enum
{
    OFF,
    ON
}   appl_status_t;
```

Parameter

```
OFF
    Switch OFF.

ON
    Switch ON.
```

appl_hw_description_s

This struct summarizes the hardware settings.

Syntax

```
typedef struct
{
    appl_status_t microphone;
    appl_status_t speaker;
    appl_signal_t signalHookPTT;
}   appl_hw_description_s;
```

Parameter

```
microphone
    State of the microphone as value of type appl_status_t

speaker
    State of the speaker as value of type appl_status_t

signalHookPTT
    Output of the PTT signals to the HW interface as value of type appl_status_t:
    ON:     Signal follows PTT key.
    OFF:    Signal follows hook switch.
```

appl_hw_reset_mode_t

This type sets the boot mode.

Syntax

```
typedef enum
{
    RESET_MODE_HA5x_SDK,
    RESET_MODE_HA20X,
    RESET_MODE_HA400
}   appl_hw_reset_mode_t;
```

Parameter
RESET_MODE_HA5x_SDK Revert to current application. RESET_MODE_HA20X Revert to HA20x emulation RESET_MODE_HA400 Revert to HA400 emulation

Functions

The following HARDWARE functions are provided:

appl_HW_switchMicro()
appl_HW_switchLP()
appl_HW_switchAudio()
appl_HW_setConfig()
appl_HW_getConfig()
appl_HW_setVolLP()
appl_HW_getVolLP()
appl_HW_setGainMic()
appl_HW_getGainMic()
appl_HW_reset()
appl_HW_getSerialNumber()
appl_HW_setFactoryResetMode()

appl_HW_switchMicro()

This function turns any microphone ON or OFF.

Syntax

```
void appl_HW_switchMicro(appl_status_t sw);
```

Parameter

sw
    State of the microphone as value of appl_status_t.

Example

```
appl_HW_switchMicro(ON);
```

appl_HW_switchLP()

This function turns any speaker/headset ON or OFF.

Syntax

```
void appl_HW_switchLP(appl_status_t sw);
```

Parameter

sw
    State of the speaker as value of typeappl_status_t.

Example

```
appl_HW_switchLP(OFF);
```

appl_HW_switchAudio()

Both, microphone and speaker are switched ON or OFF.

Syntax

```
void appl_HW_switchAudio(appl_status_t sw);
```

Parameter

sw
    State of the audio amplifier as value of appl_status_t.

Example

```
appl_HW_switchAudio(ON);
```

appl_HW_setConfig()

This function sets the hardware configuration. Typically, this function is invoked only once during the initialization of the application.

Syntax

```
void appl_HW_setConfig(appl_hw_description_s *hwc);
```

Parameter

hwc
    Pointer to the configuration struct appl_hw_description_s.

Example

```
appl_hw_description_s hwDescr;

hwDescr.signalHookPTT = SIGNAL_PTT;
hwDescr.microphone = OFF;
hwDescr.speaker = OFF;

appl_HW_setConfig(&hwDescr);
```

appl_HW_getConfig()

This function queries the hardware settings of the handset.

Syntax

```
void appl_HW_getConfig(appl_hw_description_s *hwc);
```

Parameter

```
hwc
    Pointer to the configuration struct appl_hw_description_s to be filled in.
```

Example

```
appl_hw_description_s hwDescr;
appl_HW_getConfig(&hwDescr);
```

appl_HW_setVolLP()

This function sets the speaker volume. Nine (9) volume levels are available with a range of 0 to MAX_VOLIDX_LP (8). The actual amplification associated with each volume level is documented in the User Manual.

Syntax

```
void appl_HW_setVolLP(uint8_t volIndex);
```

Parameter

```
volIndex
    Index of the volume level in a range of:
    0               Minimum value
    MAX_VOLIDX_LP   Maximum value
```

Example

```
appl_HW_setVolLP(5);
```

appl_HW_getVolLP()

This function queries the current volume level. The value returned is a level index from the range 0 to MAX_VOLIDX_LP (8). The actual amplification associated with each volume level is documented in the User Manual.

Syntax

```
uint8_t appl_HW_getVolLP(void);
```

Reply

```
uint8_t
    Current index of the volume level in a range of:
    0               Minimum value
    MAX_VOLIDX_LP   Maximum value
```

Example

```
uint8_t volume;
volume = appl_HW_getVolLP();
```

appl_HW_setGainMic()

This function configures the microphone gain. Ten (10) amplification levels are available with a range of 0 to MAX_GAINIDX_MIC (9). The actual amplification associated with each gain index is documented in the User Manual.

Syntax

```
void appl_HW_setGainMic(uint8_t gainIndex);
```

Parameter

```
gainIndex
    Index of amplification in a range of:
    0                   Minimum value
    MAX_GAINIDX_MIC     Maximum value
```

Example

```
appl_HW_setGainMic(3);
```

appl_HW_getGainMic()

This function queries the current microphone amplification as a gain index from the range 0 to MAX_GAINIDX_MIC (9). The actual amplification associated with each gain index is documented in the User Manual.

Syntax

```
uint8_t appl_HW_getGainMic(void);
```

Reply

```
uint8_t
    Gain index of current mic amplification in a range of:
    0                   Minimum value
    MAX_GAINIDX_MIC     Maximum value
```

Example

```
uint8_t gain;
gain = appl_HW_getGainMic();
```

appl_HW_reset()

This function forces a hardware reset. The handset acts like after switching on the operating voltage with a forced switch on.

Syntax

```
void appl_HW_reset(void);
```

Example

```
appl_HW_reset();
```

appl_HW_getSerialNumber()

This function returns a pointer to the serial number of the HA57. The serial number is represented by an ASCII string of a maximum of 12 characters.

Syntax

```
const char * appl_HW_getSerialNumber(void);
```

Reply

```
const char *
    Pointer to the serial number
```

Example

```
char serialNumber[15];
strcpy(serialNumber, appl_HW_getSerialNumber());
```

appl_HW_setFactoryResetMode()

This function defines how the handsets reboots after a factory reset conducted from the menu. The function is used in the init function of an application.

Syntax

```
void appl_HW_setFactoryResetMode( appl_hw_reset_mode_t mode);
```

Parameter

```
appl_hw_reset_mode_t
    Boot mode
```

Example

```
appl_HW_setFactoryResetMode( RESET_MODE_HA5x_SDK );
```

appl_Audio_largeDistance()

This function returns the state of the distance sensor, normally used to reduce the audio volume.

Syntax

```
appl_status_t appl_Audio_largeDistance(void);
```

Result

```
appl_status_t
    ON free sensor
    OFF covered sensor
```

Example

```
bool covered = OFF == appl_Audio_largeDistance();
```

appl_Audio_capabilitiesRouting()

This function returns a mask of supported audio routing hardware. The result depends on hardware variants.

Syntax

```
uint16_t appl_Audio_capabilitiesRouting(void);
```

Result

```
uint16_t (bit mask)
    AUDIO_INTERN internal speaker and microphone
    AUDIO_HEADSET headset support
    AUDIO_HEADSET_PLUGGED headset plug notification support
    AUDIO_PRIVATE audio for direct on ear usage
    AUDIO_FREE_HANDS audio for free speaking
```

Example

```
if (!(AUDIO_HEADSET_PLUGGED & appl_Audio_capabilitiesRouting())) { /* menu based
headset routing */ }
```

appl_Audio_getRouting()

This function returns a mask of supported audio routing hardware. The result depends on hardware variants.

Syntax

```
uint16_t appl_Audio_capabilitiesRouting(void);
```

Result

```
uint16_t (bit mask)
    AUDIO_INTERN internal speaker and microphone active
    AUDIO_HEADSET headset active
    AUDIO_HEADSET_PLUGGED headset plugged (can be inactive)
    AUDIO_PRIVATE audio for direct on ear usage active
    AUDIO_FREE_HANDS audio for free speaking active
```

Example

```
if ((AUDIO_HEADSET_PLUGGED & appl_Audio_getRouting()))
{ appl_Audio_setRouting(AUDIO_HEADSET); }
```

appl_Audio_setRouting()

This function sets the given audio routing.

Syntax

```
bool appl_Audio_setRouting(uint16_t route);
```

Parameter

```
uint16_t (bit mask)
    AUDIO_INTERN activate internal speaker and microphone
    AUDIO_HEADSET activate headset (ignores private/free hands)
    AUDIO_PRIVATE activate audio for direct on ear usage
    AUDIO_FREE_HANDS activate audio for free speaking
```

Result

```
bool
    true the requested route becomes active
    false hardware problem, e.g. the requested route or the mask combination is not
supported.
```

Example

```
appl_Audio_setRouting(AUDIO_INTERN | AUDIO_FREE_HANDS);
```

appl_Audio_dtmfKeyClicks()

This function sets the given audio routing.

Syntax

```
bool appl_Audio_dtmfKeyClicks(bool on_off);
```

Parameter

```
bool on_off
    true activate DTMF key tones for keys 0-9, * and # if supported by hardware
    false deactive DTMF key tones
```

Result

```
bool
    true if the request is supported
    false if the request is not supported by hardware
```

Example

```
appl_Audio_dtmfKeyClicks(true);
```

appl_Audio_beepHz()

This function creates a dual tone if supported by hardware (HA40 2x sinus, can create any DTMF tone combination). The route of the tone can be controlled by the system menu.

Syntax

```
void appl_Audio_beepHz(unsigned duration_ms, unsigned freq1_Hz, unsigned freq2_Hz);
```

Parameter

```
duration_ms
    tone duration in miliseconds
freq1_Hz
    frequency 1
freq2_Hz
    frequency 2
```

Example

```
appl_Audio_beepHz(10000, 1000, 0); // 1kHz sinus test tone for 10 seconds on HA40
```

appl_HW_powerOff()

This function sets the power parameter and initiates a power off event.

Syntax

```
void appl_HW_powerOff(uint8_t wakeupMask);
```

Parameter

```
wakeupMask
    0 deactivate ON/OFF support of the device (red key cannot switch off anymore)
    WAKEUP_POWER_CYCLE start on electric power cycle
    WAKEUP_POWER_BUTTON start on red key (always an implicit activation)
    WAKEUP_EMGY_BUTTON start on emergency key
    WAKEUP_SERIAL_EVENT start on serial traffic
```

Example

```
appl_HW_powerOff(WAKEUP_POWER_CYCLE | WAKEUP_SERIAL);
```

appl_HW_powerOn()

This function returns the power on reason and power capabilities for hardware variants (HA40 supports all).

Syntax

```
void appl_HW_powerOn(uint8_t &caps);
```

Parameter

```
caps (out parameter, can be NULL)
    WAKEUP_POWER_CYCLE start on electric power cycle

    WAKEUP_POWER_BUTTON start on red key
    WAKEUP_EMGY_BUTTON start on emergency key
    WAKEUP_SERIAL_EVENT start on serial traffic
```

Result

```
WAKEUP_POWER_CYCLE start on electric power cycle
WAKEUP_POWER_BUTTON start on red key (always an implicit activation)
WAKEUP_EMGY_BUTTON start on emergency key
WAKEUP_SERIAL_EVENT start on serial traffic
WAKEUP_OTHER alternative start, e.g. by watchdog, hardware error, forced reboot
```

Example

```
if (WAKEUP_SERIAL_EVENT & appl_HW_powerOn(NULL)) { /* start serial protocol */ }
```

Functions of the C Standard Library

The runtime system provides implementations of many functions of the C standard library. Using these implementations avoids the linking of separate library implementations into the application, which keeps the overall memory footprint of the application small.

In general, the provided functions match the behaviour of their corresponding standard implementations. Therefore, this section will only detail deviating specifics. For a general reference of the C library functions (parameters and examples), refer to the literature about the C programming language.

appl_c_itoa()

This function converts an integer into a zero-terminated string representation. This function is not based on an equivalent standard C library function.

Syntax

```
char *appl_c_itoa(int n, char *valueStr);
```

Parameter

```
n
    Integer to convert into a string representation.

valueStr
    Pointer to the result buffer.
```

Reply

```
char *
    A pointer to the result buffer for a successful conversion, NULL otherwise.
```

Example

```
char buffer[16];
int  n = 20;

appl_c_itoa(n, buffer);
```

appl_c_malloc()

This function allocates memory dynamically. This function matches the standard C function *malloc()*.

> **Notice**
>
> This is firmware memory, which is extremely limited in the heap/stack. Dynamic memory usage in embedded environments' easy fragments. It is not recommended to use this possibility! The actual heap size is

Syntax

```
void *appl_c_malloc(u32 s);
```

appl_c_free()

This function releases memory previously allocated dynamically. This function matches the standard function *free()*.

Syntax

```
void appl_c_free(void *);
```

appl_c_atoi()

This function converts a numerical string into an integer. This function matches the standard C function *atoi()*.

Syntax

```
int appl_c_atoi(const char *valueStr);
```

appl_c_hexToBin()

This function converts a hexadecimal digit into its numeric value.

Syntax

```
uint8_t appl_c_hexToBin(char h);
```

Parameter

```
h
    ASCII character representing a hexadecimal digit ('0'...'9', 'a'...'f', 'A'...'F'
)
```

Reply

```
uint8_t
    Numeric value 0x0...0xF of successfully converted hex digit, 0x80 otherwise.
```

Example

```
char c = 'E';
uint8_t   b;

b = appl_c_hexToBin(c);

// Result:  b = 0x0E
```

appl_c_strlen()

This function determines the length of a string. This function matches the standard C function *strlen()*.

Syntax

```
unsigned int appl_c_strlen(char *str);
```

appl_c_strncmp()

This function compares two strings with limitation of length. This function matches the standard C function *strncmp()*.

Syntax

```
int appl_c_strncmp(const char *s1, const char *s2, unsigned int n);
```

appl_c_strcmp()

This function compares two strings. This function matches the standard C function *strcmp().*

Syntax

```
int appl_c_strcmp(const char *s1, const char *s2);
```

appl_c_strcpy()

This function copies a string into another string. This function matches the standard C function *strcpy()*.

Syntax

```
char *appl_c_strcpy(char *s1, const char *s2);
```

appl_c_strncpy()

This function copies two strings with limitation of length. This function matches the standard C function *strncpy()*.

Syntax

```
char *appl_c_strncpy(char *s1, const char *s2, unsigned int n);
```

appl_c_strcat()

This function appends a string to another one. This function matches the standard C function *strcat()*.

Syntax

```
char *appl_c_strcat(char *s1, const char *s2);
```

appl_c_strchr()

This function determines the position of a character within a string. This function matches the standard C function *strchr()*.

Syntax

```
char *appl_c_strchr(const char *s, char c);
```

appl_c_memcmp()

This function compares two memory areas. This function matches the standard C function *memcmp()*.

Syntax

```
int appl_c_memcmp(const void *s1, const void *s2, unsigned int n);
```

appl_c_memcpy()

This function copies the content of a memory area. This function matches the standard C function *memcpy()*.

Syntax

```
void *appl_c_memcpy(void *s1, const void *s2, unsigned int n);
```

appl_c_memchr()

This function searches the first *n* Bytes of a buffer *s* for the value *val*. This function matches the standard C function *memchr()*.

Syntax

```
void *appl_c_memchr(void *s,  unsigned char val, unsigned int n);
```

appl_c_memmove()

This function moves memory areas. This function matches the standard C function *memmove()*.

Syntax

```
void *appl_c_memmove(void *s1, const void *s2, unsigned int n);
```

appl_c_memset()

This function initializes a memory area with a defined byte value. This function matches the standard C function *memset()*.

Syntax

```
void *appl_c_memset(void *s, unsigned char val, unsigned int n);
```

appl_c_sprintf()

This function performs formatted output to a string buffer. This function is a limited derivation of the standard function *sprintf() mapping* to KEIL sprintf.

Note: Functions with variable arguments can be limited in number of parameters and/or used stack size. Different compiler can produce different results. The functions should be checked within the used compilation environment before using in production.

Syntax

```
int appl_c_sprintf(char *buf, const char *fmt, ...);
```

Parameter

```
buf
    Pointer to a buffer where the resulting string is stored.

fmt
    Format string specified the same way as for the standard function.

...
    Arguments of the format string. This implementation accepts at most two arguments.
```

Reply

```
int
    As defined for the standard function.
```

Example

```
char buffer[16];
int  n = 20;

appl_c_printf(buffer,"Value n: %d\r\n",n);
```

appl_c_snprintf()

Output formatting. This function is a derivation of *snprintf() mapping* to KEIL snprintf.

Syntax

```
int appl_c_snprintf(char *buf, size_t n, const char *fmt, ...);
```

Parameter

buf
    Points to a buffer where the resulting string is saved.

n
    Length of the output field.

fmt
    Format string, equals the standard function.

...
Arguments of the format string. In this case, only two arguments are permitted.

Reply

int
    The return value equals the standard function.

Example

```
char buffer[16];
int  n = 20;

appl_c_printf(buffer,16,"Value n: %d\r\n",n);
```

appl_c_vsnprintf()

Output formatting. This function is a derivation of *vsnprintf() mapping* to KEIL vsnprintf.

Syntax

```
int appl_c_vsnprintf(char *buf, size_t n, const char *fmt, va_list ap);
```

Parameter

```
buf
    Pointer to the buffer in which the resulting string is stored.

n
    Length of the output field.

fmt
    Format string, equals the standard function.

ap
    Argument list.
```

Reply

```
int
    The return value equals the standard function.
```

Example

```
char buffer[16];
int  n = 20;
va_list args;
va_start ( args, format );

appl_c_vsnprintf(buffer,16,"Value n: %d\r\n",args);
va_end (args );
```

appl_c_sscranf()

Input formatting. This function is a derivation of sscanf() *mapping* to KEIL sscanf.

Syntax

```
int appl_c_sscanf(const char *buf, const char *fmt, ...);
```

Parameter

```
buf
    Pointer to the buffer in which the resulting string is stored.

fmt
    Format string, equals the standard function.
```

Reply

```
int
    The return value equals the standard function.
```

Example

```
char buffer[16] = "Value n: 42";
int  n;

appl_c_vsscanf(buffer,"Value n: %d", &n);
```

Software

appl_SW_getVersion()

This function returns the software version as a string. For versions tests, it is recommended only to relate to the main version, if necessary, as well to the subversion, but never relate to the build (patch) version.

Syntax

```
Syntax:    char *appl_SW_getVersion( void );
```

Reply

```
char *
    Points to the version string in major.minor.build format.
        Major: main version
        Minor: subversion
        Build: a build-calculator depending consecutive number.
```

Example

```
char buffer[6];

appl_c_strncpy (buffer, appl_SW_getVersion(), sizeof(buffer));
```

Menu Control

The following functions control some specific menu items:

appl_MENU_setEmulation()
appl_MENU_setFactoryReset()
appl_MENU_setSetupMode()

appl_MENU_setEmulation()

The menu item "Emulation" of the HA40 setup is displayed or hidden. The default setting is TRUE.

Syntax

```
void appl_MENU_setEmulation( bool enable );
```

Parameter

```
bool
    TRUE  the menu item is displayed
    FALSE the menu item is hidden
```

Example

```
appl_MENU_setEmulation(FALSE);
```

appl_MENU_setFactoryReset()

The menu item "factory reset" of the HA40 setup is displayed or hidden. The default setting is TRUE.

Syntax

```
void appl_MENU_setFactoryReset ( bool enable );
```

Parameter

```
bool
    TRUE  the menu item is displayed
    FALSE the menu item is hidden
```

Example

```
appl_MENU_setFactoryReset (FALSE);
```

appl_MENU_setSetupMode()

Use this function to prevent or allow the access to the setup menu of the HA40. The default setting is TRUE.

Syntax

```
Syntax:    void appl_MENU_setSetupMode ( const bool enable );
```

Parameter

```
bool
    TRUE   the setup menu is available
    FALSE accessing the setup menu is not possible
```

Example

```
appl_MENU_setSetupMode (FALSE);
```

Animations

For displaying activities, two animations are available.

appl_drawAnimation()
appl_drawProgress()

appl_drawAnimation()

A cyclic pulsating circle, intended as loading animation is displayed. The animation overwrites the available display content and needs a stop command to be overwritten.

Syntax

```
void appl_drawAnimation( appl_status_t status );
```

Parameter

```
status
    OFF     the animation stops
    ON      the animation starts
```

Example

```
appl_drawAnimation(ON);
```

appl_drawProgress()

A progress bar is displayed which can be used for visualizing finite processes. The progress bar overwrites available display content. The outline of the bar is displayed with a degree of filling ranging from 0 to 100 %. To remove the progress bar, the display needs to be cleared.

Syntax

```
void appl_drawProgress( unsigned int level );
```

Parameter

```
level
    0 ... 100
```

Example

```
appl_drawProgress( 50 );
```

Font Extension

The display of texts of the HA40 bases on the freely available Google fonts RobotoMono (https://fonts.google.com/specimen/Roboto+Mono). The font description was translated with a converter (https://littlevgl.com/ttf-font-to-c-array) into C code and integrated into the firmware. For each character to be shown in the display, the character is translated into a UTF-32 character. If active, the font extension of the application is called. Without a result, the internal bitmap is determined and respectively shown. If the font extension has a result, it is used. In this way, applications can extend internal character sets as well as overwrite them.

The font extension function

Syntax

```
void char_info(uint32_t utf32, appl_typeface_t font, appl_char_info_t *info);
```

has the UTF-32 character as parameter, the font to be displayed and a pre-filled (font_width and font_height, rest 0) character information. The struct must be completed, if width isn't 0, the character is displayed as given. The missing bitmap realizes a blank of the given width.

Syntax

```
typedef struct
{   uint8_t font_width;
    uint8_t font_height;
    uint8_t height;
    uint8_t width;
    const uint8_t* bitmap;
    uint8_t alias;  // new since SDK API 4
}   appl_char_info_t;
```

Parameters

```
font_width:
    Number of horizontal pixels for a character, standard for monospace fonts.

font_height:
    Number of vertical pixels of a font.

height:
    Height of the bitmap in pixels.

width:
    Width of a given bitmap in pixels (left aligned rounded to byte).

bitmap:
    The byte array of the bitmap for the character (can be 0).


alias:
    default 1, supported 2 or 4, number of bits per pixel
```

It is permitted that height and width may differ from those specified. The character is always displayed completely with the given bitmap at the respective position, meaning that it might overwrite neighbouring characters. In reverse, characters that are too wide can be overwritten later with other characters. Normally, this can only be noticed with monospace fonts, which are used quite intensely by normal protocol applications.

The SDK of the HA40 now offers the possibility of variable character placing (SH_*VARIABLE*) where strings are set with the true width of the character, but ligatures are not implemented.

The provided example demonstrates the firmware-internal implemented pixel fonts. If the available bitmaps are removed and extended by, for example, the Greek language, European languages are covert. If just an extension of fonts is asked for, a protocol extension without event handler may be created for the intended protocol.

If UTF-8 is used in custom applications, it is recommended to use a UTF-8 capable IDE or editor (gvim) to force, especially for Windows, the UTF-8 usage and storage.

Note: Complete UTF code pages can be precompiled by pei tel and loaded into the SPI configuration flash. Contact your distributor with the font description (with the alternative ttf font description and/or enhanced font page ranges).

END OF DOCUMENT