



Byssus API

Developer's Guide & Reference Manual

Table of Contents

1	Introduction.....	3
2	Byssus Deliverables.....	4
2.1	Installation.....	4
2.2	Directory Structure.....	4
2.3	Compiler/Linker Support.....	5
2.4	Library Identity.....	6
3	A Simple Example – Hello World.....	7
3.1	Full Source.....	7
3.2	Line-By-Line Description.....	8
4	User Interface Integration.....	9
4.1	Use Of Callbacks.....	9
4.2	Progress Callback.....	9
4.3	Alert Callback.....	11
4.4	Input Callback.....	11
4.5	Input Validation Function.....	15
4.6	Sample.....	17
5	Other API Services.....	19
5.1	Purchase Or Activation On Demand.....	19
5.2	Integration With System Management Tools.....	19
5.3	Error Reporting.....	20
5.4	Internationalisation.....	21
5.5	Multi-Currency Pricing.....	22
6	Protection Of Executables.....	24
6.1	Verification Strategies.....	24
6.2	The Case Against A Dynamic Library.....	29
7	Generation Of Executables.....	30
7.1	Compilation Process.....	30
7.2	Linking.....	30
8	Internal Runtime Operations.....	31
8.1	Licence Verification.....	31
8.2	Installation And Re-Installation Of Licences.....	31
8.3	Access To The Internet.....	31
8.4	Use Of The Local File System.....	32
9	Appendix – Function Reference.....	33
9.1	byssus_activate_licences.....	34
9.2	byssus_buy_licences.....	35
9.3	byssus_check_licensed.....	37
9.4	byssus_get_pricelist.....	39
9.5	byssus_get_stats.....	40
9.6	byssus_init.....	41
9.7	byssus_load_licences.....	43
9.8	byssus_strerror.....	44
9.9	byssus_validate_form.....	45

1 Introduction

Thank you for using Byssus!

This manual explains how to use the Byssus API to add licensing to your applications and to integrate them with the Byssus service.

This manual requires knowledge of the C language and of its compilation/link process. An understanding of network communication over the Internet and of graphical user interface concepts is useful, but not essential.

We first start this manual with an overview of what constitutes the Byssus Software Development Kit (SDK). We then present a simple example program using the API.

A detailed description of each service provided by the API follows. As you will see, the Byssus API was designed to be as easy to use as possible. The API contains only a few function calls, all of which are fully detailed. Several examples are also included to support the description of those functions or the simple data structures they use.

Background information about how to protect an application is also included. This manual concludes with instructions on how to compile and link with the Byssus library and with details about how the Byssus library works internally.

The appendix serves as reference manual for all Byssus functions.

2 Byssus Deliverables

2.1 Installation

You should have received a custom archive file from Byssus. To install the Byssus SDK and accompanying files, just decompress this archive file where it will suit your development needs.

There is no restriction on installing the archive in a shared directory.

2.2 Directory Structure

The compressed archive file you received from Byssus contains all the necessary files needed to integrate your application with the Byssus service. These files are in five directories:

- documentation
- headers
- samples
- internationalisation
- libraries

The content of the entire directory hierarchy is presented in this table:

Pathname:	Description:
README	Instructions on where to find this manual in the Byssus package
docs	Directory for all documentation files
docs/api.pdf	This manual
headers	Directory for all header files
headers/byssus	Directory containing all Byssus-specific header files
headers/byssus/api.h	Main #include file. Other headers below are included by this one
headers/byssus/countries.h	Definition of country names and manipulation types
headers/byssus/errors.h	Definition of error codes and messages
headers/byssus/fields.h	Definition of all input fields and related types
headers/byssus/progress.h	Definition of progress messages
samples	Directory containing all demonstration source files
samples/hello.c	Simple demonstration program
samples/game.c	More complete demonstration program

Pathname:	Description:
samples/activate.c	Simple activation utility used for deployments
samples/ui_integration.c	Declaration of callback functions shared by all sample programs
samples/ui_integration.h	Header associated with the file above
i18n ¹	Directory containing all text strings used by the Byssus library that can be localised
i18n/card_names.c	Payment card names in English
i18n/country_names.c	Country names in English
i18n/error_messages.c	Byssus error messages in English
i18n/progress_messages.c	Byssus status reports in English
i18n/currencies.c	Application price list in several currencies
lib/	Directory containing Byssus binaries
lib/byssus.lib ²	The custom library you need to statically link with your application

2.3 Compiler/Linker Support

The Byssus API presents a pure C interface to the linker. As such, the library can be natively linked with other C/C++ libraries, as well as with code developed in other programming languages.

2.3.1 C/C++ Development Environments

Byssus directly supports all mainstream C/C++ development environments:

- Microsoft Visual C++
- Borland C++ Builder
- GNU C/C++

OpenWatcom C/C++ is supported only indirectly at the moment³, relying on its object file format compatibility with Microsoft's linker.

2.3.2 Other Programming Languages

The Byssus library can be linked with code developed in other programming languages as long as:

-
- 1 i18n is short for "Internationalisation".
 - 2 The exact file name of this library may vary depending on which compiler you requested the Byssus library to be generated for.
 - 3 Byssus may support OpenWatcom directly in the future if clients require it.

- The other language's compiler provides a mapping mechanism with C libraries.
- The linker used is compatible with one of the object file formats used by the C/C++ environments listed in the previous section.

The Byssus library should not be used with interpreted⁴ languages.

2.4 Library Identity

All libraries (“byssus.lib”) generated by Byssus are functionally identical. However, each library is in fact custom, and contains several identifiers making each library unique and *bespoke*. These identifiers⁵ include:

- Vendor name
- Product name
- Product version
- Package ID
- Encryption information
- Library version

The Byssus server uses some of this information to customise licence management according to your requirements. It is therefore recommended that you use a new library (provided at your request as part of Byssus' ongoing service) every time you release a new version of your applications.

4 Although this is technically possible, this is NOT recommended. See section 2.4.

5 These identifiers are not accessible from the API and are used internally by the library when communicating with the Byssus server.

3 A Simple Example – Hello World

The Byssus service makes it impossible for users to illegally use software past the end of its trial period. The Byssus service is also extremely secure and uses encryption mechanisms superior to those currently provided by mainstream operating systems and Internet applications.

To deliver the added-value, power and security of the Byssus service to developers, the Byssus API was designed with three main goals:

- Applications using the Byssus service retain their ease of use for end users.
- Developers can easily integrate the Byssus service into their applications.
- Applications using the Byssus API are extremely difficult to hack.

The following example shows how easy it is to use the Byssus API. Later sections in this document will focus on user-friendliness and barriers to hackers.

3.1 Full Source

Now let's see how a simple *Hello World* program could be licensed. The code below includes a header declaring all callbacks⁶ passed to `byssus_init()` as well as error checking instructions. It is fully compilable:

```
#include <stdio.h>
#include <byssus/api.h>

#include "user_interface_integration.h" // callbacks

int main()
{
    byssus_error e;

    // Initialise Byssus API
    e = byssus_init(collect_data, show_progress, show_alert);
    if (e) panic(e);

    // Load licences in memory
    e = byssus_load_licences();
    if (e) panic(e);

    // Check licences
    e = byssus_check_licensed(100, 'A');
    if (e) panic(e);

    // Show message
    printf("Hello, World !\n...Licensed using Byssus.\n");

    // Return to shell
    return 0;
}
```

This source file can be found in the *samples* directory.

⁶ The callback mechanism is presented in section 4.1 and sample source is shown in section 4.6.

3.2 Line-By-Line Description

The Byssus API requires only one header file to be included in your code:

```
#include <byssus/api.h>
```

The Byssus API must first of all be initialised:

```
byssus_init(show_progress, collect_data, show_alert);
```

The parameters passed to `byssus_init()` will be detailed later. Broadly speaking, these are function callbacks to allow users to enter their payment details.

Before entering the main body of the application, licence⁷ codes must be loaded into memory. This must be done only once by calling:

```
byssus_load_licences();
```

The exact mechanisms used to load licence codes will be described later in section 8.1.

Finally, licences must be checked from at least one place in the application to authorise execution:

```
byssus_check_licensed(100, 'A');
```

The parameters passed to `byssus_check_licensed()` will be explained later in section 6.1.

That's it! Just one header file and three function calls. Although there are other functions in the API, you have been presented with how the API must generally be used. It's that simple.

⁷ Either temporary licences or paid licences.

4 User Interface Integration

The previous chapter introduced how to use the Byssus API to license the execution of an application. This chapter presents the other facet of the Byssus API: the interface with the user, including the collection of payment information.

Note *The Byssus API is not limited to Graphical User Interfaces (GUI) and is designed to work just as well with text-based interfaces. However, the rest of this document assumes that the Byssus API is integrated with an application using a GUI, and consequently uses terms which are more relevant to GUIs than to text-based interfaces.*

Note *We avoid using terms specific to any particular GUI toolkit in order for this document to be understood by the broadest audience, which might not be familiar with any toolkit we might have chosen. The Byssus API can be used with the GUI toolkit of your choice: Microsoft, Borland, GTK+ or any other.*

4.1 Use Of Callbacks

A rapid definition first. A callback is a function whose signature is defined by a library, but which is implemented by the user of that library. The library has full control over when the function will be executed, if at all.

The Byssus API makes use of callback functions to interface with the user of the application. More precisely, the Byssus API requires callbacks to be implemented in order to collect payment details from the end-user and to report progress status or alerts to the end-user.

The Byssus API defines three callback function types which are presented in the next sections. Their implementation requirements are designed to be simple, in order to ensure their rapid and easy implementation. As seen in the *Hello World* source, these callbacks must be provided to function `byssus_init()`.

4.2 Progress Callback

The first callback type is used by `byssus_load_licences()` when loading a temporary licence from the Byssus server on the Internet as well as by `byssus_check_licensed()` when requesting permanent licences from the Byssus server. As this operation can take a few seconds, good user interface design guidelines require application developers to inform the user what operations are taking place. This first Byssus callback is designed to do just that.

The prototype of this callback function is:

```
typedef bool (* byssus_show_progress) (
    byssus_progress_step current_step,
    byssus_progress_step max_step);
```

When called, a typical implementation of this function callback will open a modal dialog⁸ containing a progress bar and update that bar every time the function is called (by the Byssus API) until parameters `current_step` and `max_step` are equal. At which point, the application should close the dialog.

The Byssus API guarantees that parameter `current_step` will always be equal to `BYSSUS_PROGRESS_INITIALISE` the first time and to `BYSSUS_PROGRESS_DONE` the last time. It also guarantees that values for this variable will increment with every call, but it does not guarantee that the increment will be one.

Implementations of this callback also commonly include a “Cancel” button inside the dialog to allow end-users to cancel the temporary licence request. Returning `false` will indicate to the Byssus API that the operation must be aborted.

Note *When credit card payment is confirmed by the bank it will no longer be possible to cancel the overall operation. In this case, even when the callback function returns false, the overall operation will continue and the function invoking the callback will return successfully.*

Alongside the progress bar, the application may also want to display a description of the steps taken by the Byssus API. Messages corresponding to values of type `byssus_progress_step` are made available by the API:

```
const char * byssus_progress_messages[];
```

The use of this array with this callback is shown below.

4.2.1 Sample

In a simple text-based application, a progress callback could be implemented as follows:

```
bool show_progress(
    byssus_progress_step current,
    byssus_progress_step limit)
{
    bool clicked_on_cancel;

    if (current == BYSSUS_PROGRESS_INITIALISE) {
        printf("Attempting to obtain licences ...\n");
    }

    printf("Step %d of %d: %s\n", current, limit,
        byssus_progress_messages[current]);

    clicked_on_cancel = false; // simulate user NOT clicking "Cancel"

    if (clicked_on_cancel || current == BYSSUS_PROGRESS_DONE) {
        printf("Done.\n");
    }
}
```

⁸ A dialog is considered modal when the user can't get to other windows or controls in the application.

```

        return ! clicked_on_cancel;
    }

```

This source, like all other sample callbacks can be found in the file “samples/user_interface_integration.c”.

4.3 Alert Callback

This second callback is needed by function `byssus_check_licensed()` to alert the user of the application to an error, including the lack of any valid licences. The prototype of this callback function is very simple:

```

typedef void (* byssus_show_alert)(
    const char * message);

```

This function takes a C string as parameter and is expected to display the message to the end-user. A typical implementation will open a modal dialog to display the error message, and include an “OK” button inside the dialog to allow the user to dismiss the alert.

4.3.1 Sample

At a bare minimum, such a callback function could be implemented like this:

```

void show_alert(const char * msg)
{
    printf("Alert: %s\n", msg);
}

```

4.4 Input Callback

The goal of the third callback function is to allow the end-user to obtain permanent licences to be able to continue executing an application on his computer after the end of the trial period. This callback is used by function `byssus_check_licensed()`.

From a functional point of view, the end-user must be presented with a form. After filling that form and confirming he wants to buy, the Byssus API will send all relevant information to the Byssus server through a secure encrypted channel. After validating the information, the Byssus server will generate the appropriate licence keys and these licences will be stored on the user's local disk automatically.

The prototype of this function is:

```

bool (* byssus_input_form)(
    byssus_form * data);

```

With type `byssus_form` defined as:

```

typedef struct {
    bool                is_payment_form;
    byssus_form_payment payment;
    byssus_form_activation activation;
} byssus_form;

```

The `byssus_form` structure allows end-users not only to purchase licences to use an application on their computer, but also to activate pre-paid licences. Pre-paid licences can be used in several contexts to allow:

- a company to buy a bulk number of applications for their staff, which then have to enter an activation code provided by Byssus to the company buyer.
- a person to buy an application on one computer but to target the execution of this application on a different computer - parents buying an application for their child, for instance.

A typical implementation of this callback will open a dialog presenting the user with a modal Wizard-type dialog. One fork of the wizard should allow the user to enter payment information as described by type `byssus_form_payment`, while the other fork must allow the end-user to enter the activation information as described in type `byssus_form_activation`.

The design of the `byssus_form` type allows the end-user to travel back and forth on both forks (using “Previous” and “Next” buttons inside a dialog) without losing previously entered information.

Field `is_payment_form` must be set to `true` to mean payment or `false` to mean activation.

The implementation should also include two buttons, “OK” to confirm the purchase, or “Cancel” in case the end-user decides not to purchase the application right now.

If the user clicks on “OK”, the application must make use of a function that validates a user's input. This function is introduced later in section 4.5.

In the next three sub-sections, we describe fields related to purchases and included in a structure of type `byssus_form_payment`. These input fields are classified as: mandatory, optional, and custom. The final sub-section presents the fields in `byssus_form_activation` related to pre-paid activations.

4.4.1 Mandatory Payment Fields

All fields directly related to payment processing are mandatory and must be provided by the user, or in one case, calculated by the application. Without this information, the API will reject the payment request without even contacting the Byssus server.

All mandatory fields are introduced here:

Field types:	Description:
Debit/Credit card information	All card details, including: the card type and number, the validity date(s), the issue number (when applicable), the name of the owner, and the security code.
Currency used for payment	Byssus currently supports US Dollars, Euros and British Pounds.
Quantity to be purchased	1 by default, but can be more.
Whether the buyer is the end-user or not	True by default. If false, the buyer will receive an activation code which will have to be used by the end-user.

Field types:	Description:
Buyer's reference (business end-users)	For corporate purchases, Byssus requires a reference number from the purchaser to support his company's business processes (e.g. purchase order number).
E-mail address	The address to which Byssus will send the invoice once payment is processed successfully, as well as activation code(s) if applicable.
Country	This field is checked by banks against the country where the card was issued. It is also needed to calculate the next field listed (VAT amount).
Tax rate	There are several possible values based on your country location, and that of your customer. Byssus, as part of its service, will help you determine what rates need to be applied to transactions.

The corresponding C definitions of these mandatory fields comprise a few C strings and a limited number of simple enumerated or structured types:

```

char          buyer_reference[BYSSUS_FIELD_LENGTH_BUYER_REFERENCE
                           +1];
char          card_number[BYSSUS_FIELD_LENGTH_CARD_NUMBER +1];
byssus_card_type card_type;
unsigned int  country_index;
byssus_currency currency;
char         email[BYSSUS_FIELD_LENGTH_EMAIL_ADRESS +1];
byssus_card_date expiry_date;
char         holder_name[BYSSUS_FIELD_LENGTH_CARD_HOLDER_NAME +1];
unsigned char issue_number;
unsigned int  quantity;
char         security_code[BYSSUS_FIELD_LENGTH_SECURITY_CODE +1];
bool         target_local_computer;
byssus_card_date valid_from;
float        tax_rate;

```

Custom C types used in some of the fields above are defined as:

```

typedef struct {
    unsigned char month;
    unsigned char year; // 2 digits
} byssus_card_date;

typedef enum {
    BYSSUS_CARD_NONE_SELECTED,
    BYSSUS_CARD_VISA,
    BYSSUS_CARD_MASTERCARD,
    BYSSUS_CARD_SWITCH,
    BYSSUS_CARD_SOLO,
    BYSSUS_CARD_DELTA,
    BYSSUS_CARD_ALEX,
    BYSSUS_CARD_TOTAL // Should only be used as array boundary
} byssus_card_type;

typedef enum {
    BYSSUS_COUNTRY_NONE_SELECTED,
    BYSSUS_COUNTRY_AFGHANISTAN,
    BYSSUS_COUNTRY_ALBANIA,
    // ... 250 lines later ...
    BYSSUS_COUNTRY_ZAMBIA,

```

```

    BYSSUS_COUNTRY_ZIMBABWE,
    BYSSUS_COUNTRY_TOTAL // should only be used as array boundary
} byssus_country; // used to assign field "country_index"

```

Custom enumerations `byssus_card_type` and `byssus_country` are designed to be used to build pop-up menus:

- Their first constant (`BYSSUS_CARD_NONE_SELECTED`, `BYSSUS_COUNTRY_NONE_SELECTED`) is meant to designate an absence of value while their last constant (`BYSSUS_CARD_TOTAL`, `BYSSUS_COUNTRY_TOTAL`) is meant to be an upper bound to be compared against using operator “<”.
- The API provides two arrays containing the corresponding textual representations of each constant defined in `byssus_card_type` and `byssus_country`. These tables are introduced later in section 5.4.

4.4.2 Optional Payment Fields

The Byssus API also includes complementary fields you may want to collect from users for marketing or customer service reasons, or that users may want to provide to you.

As all these fields are atomic and self-explanatory, we immediately present their C definitions:

```

// Name information
char  first_name[BYSSUS_FIELD_LENGTH_FIRST_NAME +1];
char  last_name[BYSSUS_FIELD_LENGTH_FAMILY_NAME +1];
char  company_name[BYSSUS_FIELD_LENGTH_COMPANY_NAME +1];

// Address information
char  address_line_1[BYSSUS_FIELD_LENGTH_ADDRESS_LINE +1];
char  address_line_2[BYSSUS_FIELD_LENGTH_ADDRESS_LINE +1];
char  city[BYSSUS_FIELD_LENGTH_CITY +1];
char  region[BYSSUS_FIELD_LENGTH_MAIN ADMINISTRATIVE_REGION +1];
char  postcode[BYSSUS_FIELD_LENGTH_POSTCODE +1];

// Contact information
char  phone[BYSSUS_FIELD_LENGTH_PHONE_NUMBER +1];
char  fax[BYSSUS_FIELD_LENGTH_FAX_NUMBER +1];
char  website[BYSSUS_FIELD_LENGTH_WEB_SITE +1];

```

In the previous section we mentioned that field `buyer_reference` is mandatory only in the case of business buyers. The Byssus API defines a transaction to be of a business nature when the end-user provides a value for the `company_name` field. Otherwise the transaction is seen as initiated by a private individual and the `buyer_reference` field is not enforced by the API.

Note *The format of postal addresses around the world is very different for postcodes and the order of address components. The Byssus API defines fields large enough to cope with every situation, and recommends using these fields in the following hierarchical order on the screen: first line of address, second line of address, city, region, post code, country.*

4.4.3 Custom Payment Field

The Byssus API also includes a general purpose, free-form field which can be used to collect additional information for your specific requirements. This field is optional and is defined as:

```
byssus_freeform    comment;
```

With `byssus_freeform` defined as:

```
typedef struct {
    unsigned int    nb_bytes;
    char            buffer[BYSSUS_FIELD_LENGTH_FREE_FORM + 1];
} byssus_freeform;
```

This structure defines a buffer of exactly 1000 bytes, which can be used freely by the application developer to store any information. The Byssus library does not check data stored in this buffer.

Warning *Make sure you always copy real data into this field, rather than just pointers to the data you intend to submit. If you store pointers in this buffer⁹, the information sent to the Byssus server will not be what you intended to collect from end-users.*

4.4.4 Activation Field

Unlike the last three sections which covered the collection of payment information, the activation field is used to obtain a permanent licence key which was paid for earlier but not issued at the time of payment. This will be the case if the purchase was made on a different computer from the one on which the application will run (proxy purchase). Whenever the `target_local_computer` field is set to `false` or `quantity` is greater than one, the server will generate an activation code for each proxy purchase. All the activation codes are then sent by email using the mandatory `email` field.

To obtain a permanent licence key, all the user has to do is to enter an activation code into the input form of the activation fork in the input wizard dialog. No payment information needs to be collected in this case.

The activation field is defined as:

```
char authorisation_code[BYSSUS_FIELD_LENGTH_AUTHORISATION_CODE + 1];
```

Note *Activation codes are fairly long strings. It is therefore recommended to enable copy and paste operations in that dialog.*

4.5 Input Validation Function

The callback function of type `byssus_input_form` should not immediately dispose of the input dialog when the user clicks on “OK” in the input wizard. Before returning to the Byssus API, you must use the following function to validate user input, and then ensure users correct their input, if necessary:

⁹ Whether stored directly or as part of a structure.

```
bool byssus_validate_form(
    byssus_form *      fields,
    byssus_error_report * all_errors);
```

Reminder *The `byssus_validate_form()` function is not a callback! But it can be used within a callback function of type `byssus_input_form`.*

This function returns `true` if all the fields passed in the first parameter pass the validation checks. This function checks the format of each field, checks that payment card dates make logical sense, and ensures that all mandatory fields are completed. For instance, it checks that a starting date is typed for Switch cards. But this function does not check for inconsistencies, such as a postcode not corresponding to the right city or country, nor does it check Credit/Debit card details (a bank does that through the Byssus server at a later stage).

If errors are detected, the function returns `false` and the caller (the callback function) must interpret the `all_errors` parameter and force the user to correct his entries. The type of this parameter is defined as:

```
typedef struct {
    byssus_error errors[BYSSUS_FIELD_ID_TOTAL];
} byssus_error_report;
```

Constant `BYSSUS_FIELD_ID_TOTAL` is defined as part of the following enumeration identifying each field in the the input form:

```
typedef enum {
    // Name
    BYSSUS_FIELD_ID_FIRST_NAME,
    BYSSUS_FIELD_ID_FAMILY_NAME,
    BYSSUS_FIELD_ID_COMPANY_NAME,

    // Address
    BYSSUS_FIELD_ID_ADDRESS_LINE_1,
    BYSSUS_FIELD_ID_ADDRESS_LINE_2,
    BYSSUS_FIELD_ID_CITY,
    BYSSUS_FIELD_ID_REGION,
    BYSSUS_FIELD_ID_POST_CODE,
    BYSSUS_FIELD_ID_COUNTRY_INDEX,

    // Contact
    BYSSUS_FIELD_ID_PHONE,
    BYSSUS_FIELD_ID_FAX,
    BYSSUS_FIELD_ID_EMAIL,
    BYSSUS_FIELD_ID_WEB_SITE,

    // Means of payment
    BYSSUS_FIELD_ID_CARD_TYPE,
    BYSSUS_FIELD_ID_CARD_NUMBER,
    BYSSUS_FIELD_ID_CARD_VALID FROM,
    BYSSUS_FIELD_ID_CARD_DATE EXPIRY,
    BYSSUS_FIELD_ID_CARD_ISSUE NUMBER,
    BYSSUS_FIELD_ID_CARD HOLDER NAME,
    BYSSUS_FIELD_ID_CARD_SECURITY_CODE,

    // Price
    BYSSUS_FIELD_ID_QUANTITY,
    BYSSUS_FIELD_ID_TARGET LOCAL COMPUTER,
    BYSSUS_FIELD_ID_CURRENCY,
    BYSSUS_FIELD_ID_BUYER_REFERENCE,

    // Computed by the client application
```

```

    BYSSUS_FIELD_ID_TAX_RATE,
    BYSSUS_FIELD_ID_COMMENT,

    // Pre-payment
    BYSSUS_FIELD_ID_AUTHORISATION_CODE,

    // End of enumeration
    BYSSUS_FIELD_ID_TOTAL
} byssus_field_id;

```

Function `byssus_validate_form()` reports all errors it finds in all fields of the form. When an error is detected in a field, its corresponding `error_code` in `byssus_error_report` is set to a value other than `BYSSUS_ERROR_NONE`. To assist developers, the Byssus API also provides a function to get an error message from an error code. This function is described later in section 5.3.

4.6 Sample

The sample code below shows the use of the API during payment processing. It focusses on data and their format rather than on common input instructions.

Before showing the sample code for the callback, we first define an error reporting routine:

```

void show_first_error(byssus_error_report * all_errors)
{
    int i;

    for (i = 0; i < BYSSUS_FIELD_ID_TOTAL; i++) {
        if (all_errors->errors[i] != BYSSUS_ERROR_NONE) {
            printf("Error in field %d: %s\n", i,
                byssus_strerror(all_errors->table[i]));
            break; // for
        }
    }
}

```

The sample callback is:

```

bool collect_data(byssus_form * data)
{
    bool done;
    bool clicked_ok;

    printf("Please enter your details.\n");

    done = false;
    do {
        // simulate user input for demo purposes.
        // All data below are examples.
        data->is_payment_form = true;
        data->payment.card_type = BYSSUS_CARD_VISA;
        strcpy( data->payment.card_number, "1234 1234 1234 1234");
        data->payment.expiry_date.month = 12;
        data->payment.expiry_date.year = 05;
        strcpy( data->payment.holder_name, "Mr J Smith");
        strcpy( data->payment.security_code, "123");
        data->payment.quantity = 1;
        data->payment.target_local_computer = true;
        data->payment.currency = BYSSUS_CURRENCY_DOLLAR;
        strcpy( data->payment.email, "jsmith@mydomain.com");
        data->payment.country_index = BYSSUS_COUNTRY_JAPAN;
        strcpy( data->payment.buyer_reference, "just a test");

        // Simulate user clicking on "OK" button
    } while (!done);
}

```

```

clicked_ok = true;

// Validate form
if (clicked_ok) {
    byssus_error_report all_errors;

    // determine VAT and
    // other application specific information
    data->payment.tax_rate = 17.5; // for example

    // Validate all data
    done = byssus_validate_form(data, & all_errors);
    if (! done) {
        show_first_error(& all_errors);
    }
} else {
    done = true;
}
} while (! done);

printf("Values entered by pseudo end-user.\n");

return clicked_ok;
}

```

5 Other API Services

Although the Byssus API is focussed on handling licensing and payment operations as automatically and transparently as possible, it also provides additional services which can be useful to application developers.

5.1 Purchase Or Activation On Demand

So far we have concentrated on collecting end user payments, or activating licences, when the trial period ends. Application developers need a mechanism for proxy licence purchases to be activated immediately and will probably want to offer end-users the possibility to buy before the end of the trial period.

The function to use in this case is:

```
byssus_error byssus_buy_licences();
```

It is a simple function which makes use of the same callbacks presented in the previous chapter. In fact, this function is called by `byssus_check_licensed()` internally to handle licence payment and activation.

This function works in a similar way to `byssus_load_licences()`: the function loads licences in memory (if payment or activation was successful), but performs no verification of those licences, relying instead on the application to call function `byssus_check_licensed()` later.

5.2 Integration With System Management Tools

This section addresses the end-user requirements of large organisations. In such organisations it is common for a buyer to purchase an application in bulk. Byssus supports such bulk proxy purchases and once payment is completed, provides the buyer with a list of activation codes by email.

As mentioned in section 4.4.4, the buyer could then distribute one code to each employee requiring use of the application. Each employee would then have to copy/paste this activation code into the input form wizard to link the application to the user's computer.

This could be an acceptable solution in the case of users who are IT-literate. But some users might not feel comfortable and may require help desk assistance.

The other solution is to provide an “activation” utility which can be executed by system management tools to deploy your application on multiple computers, without requiring any end-user intervention. This utility would probably be used within a script, batch or a tool such as Microsoft's SMS. Such an activation utility would take an activation code as its only parameter and would be remotely executed on every system that needed to have a permanent licence key.

This utility would make use of only two Byssus functions: `byssus_init()` and a new function:

```
byssus_error byssus_activate_licences(  
    byssus_form_activation * activation_code);
```

Important *Both the application and the activation utility must be linked with exactly the same Byssus library. Refer to section 2.4.*

5.2.1 Sample

This code shows an example of an activation utility distinct from a main application. Note the NULL parameters passed to `byssus_init()` and how an activation code is passed to `byssus_activate_licences()`.

```
int main(int argc, char * argv[])  
{  
    byssus_error e;  
    byssus_form_activation data;  
  
    // Check activation code is passed on the command line  
    if (argc != 2) {  
        return BYSSUS_ERROR_NO_VALUE;  
    }  
    if (strlen(argv[1]) >= BYSSUS_FIELD_LENGTH_AUTHORISATION_CODE) {  
        return BYSSUS_ERROR_INVALID_VALUE;  
    }  
  
    // Initialise Byssus API  
    e = byssus_init(NULL, NULL, NULL);  
    if (e) return e;  
  
    // Activate licences on a computer  
    strcpy(data.authorisation_code, argv[1]);  
    e = byssus_activate_licences(& data);  
    if (e) return e;  
  
    // Return to shell  
    return BYSSUS_ERROR_NONE;  
}
```

Warning *Function `byssus_load_licences()` depends on a user interface and consequently should not be called from such an activation utility..*

5.3 Error Reporting

Most functions in the Byssus API return an error code of type `byssus_error`. Although all error codes can be seen in the header file¹⁰, an error code is only useful if it can be turned into an error message. The Byssus API provides such a function, inspired by the `strerror()` function of the standard C library:

```
const char * byssus_strerror(byssus_error errnum);
```

This function returns a C string for any valid `byssus_error` value passed as parameter.

¹⁰ See file `<byssus/errors.h>` if needed.

5.4 Internationalisation

In this document we have so far mentioned four tables containing messages related to:

- progress messages,
- country names,
- payment card names,
- error messages.

These tables are inherently part of the Byssus API. However they are provided to you in source code format, in order to allow you to localise¹¹ those messages as well, should you need to.

All these tables can be found in the “i18n” directory.

5.4.1 Single Column Tables

We start with the simplest table format, used for progress messages. It is a basic array of C strings. The size of this array matches the number of elements in enumeration type `byssus_progress_step`:

```
extern const char * byssus_progress_messages[BYSSUS_PROGRESS_TOTAL];
```

To localise this table, you only need to replace the existing messages in English found in “`byssus_progress_messages.c`” with messages translated in the chosen target language.

5.4.2 Two Column Tables

The Byssus API defines codes to represent countries, errors or payment cards. These codes are defined through enumerations, such as the ones presented in section 4.4.1. These enumerations are convenient to define a type, regrouping constants of similar meaning, with each constant using a unique value within the enumeration.

Single-column tables are sufficient to support a single language, but do not provide adequate support for internationalisation. Byssus created its `byssus_country` enumeration by placing each country in alphabetical order in English. But each country might have a different name in other languages. For instance, Germany is written “Deutschland” in German and “Allemagne” in French.

Because we defined our enumeration in alphabetical order in English, if we used a single column table to represent country names, and wanted to display the content of that table translated into German for example, the table would be displayed as a list of unsorted names. This is not what you need if you wish to present an easy to use pop-up menu to the end-user!

This is why Byssus uses two column tables, so that tables can be displayed in alphabetical order for any language, without having to redefine the type `byssus_country`.

Such tables are used in three cases: country names, error messages and card names, and are defined like this:

¹¹ To translate those messages into a language other than English.

```
extern const
byssus_country_description    byssus_country_names[BYSSUS_COUNTRY_TOTAL];

extern const
byssus_error_description     byssus_error_messages[];

extern const
byssus_card_description      byssus_card_names[BYSSUS_CARD_TOTAL];
```

The content of these tables can be found in the “i18n” directory, but we present an example row for each table:

```
{ "Australia",                                BYSSUS_COUNTRY_AUSTRALIA }
{ "A mandatory value is missing", BYSSUS_ERROR_NO_VALUE }
{ "Visa",                                       BYSSUS_CARD_VISA }
```

As can be seen, each row in these tables contains two elements of information:

- the code as defined in an enumeration (byssus_country for instance),
- the text string corresponding to that code.

To localise such tables, an application developer only has to:

1. create a new file for the target language by copying the English version,
2. translate each text string in the table into the target language,
3. re-sort the table rows alphabetically by the translated text strings.

A country pop-up menu can be built from the localised table. The index returned from an end-user menu selection can then be used to look up the country code from the localised table.

Note *The country codes enumerated in type byssus_country do not follow International Standard ISO 3166-1. The Byssus API hides from the application developer the complexity of managing such non-sequential codes. ISO 3166-1 is however used internally by Byssus' library and server.*

5.5 Multi-Currency Pricing

The Byssus service gives end-users a choice of purchase currency. Byssus currently supports three currencies: US Dollar, Euro, British Pound¹².

Currencies are defined by an enumeration:

```
typedef enum {
    BYSSUS_CURRENCY_NONE_SELECTED,
    BYSSUS_CURRENCY_DOLLAR,
    BYSSUS_CURRENCY_EURO,
    BYSSUS_CURRENCY_POUND,
    BYSSUS_CURRENCY_TOTAL // Should only be used as a boundary
} byssus_currency;
```

As seen in section 4.4.1, end-users must select a payment currency from those available - typically through a pop-up menu. Prices in each supported currency are

¹² Byssus may in the future support additional currencies based on client needs and market demand.

retrieved from the Byssus server using the following function:

```
byssus_error byssus_get_pricelist(byssus_pricelist * pricelist);
```

Note *This function can only be used during the execution of a byssus_input_form callback.*

This function returns the following structure in the `pricelist` parameter:

```
typedef struct {  
    float prices[BYSSUS_CURRENCY_TOTAL];  
} byssus_pricelist;
```

Note *Byssus does not provide pre-formatted strings to build a pop-up menu directly. This is because the position of the currency symbol varies from country to country. For example, “\$” is placed before the amount in the USA, but “€” is placed after the amount in France. Other differences may also apply when localising the display of your price-list.*

6 Protection Of Executables

6.1 Verification Strategies

In our *Hello World* example, we quickly introduced the use of function `byssus_check_licensed()`. In this section we explain how to use it in a more refined way, to achieve higher levels of security.

6.1.1 Security and Determinism

When protecting an application, one must always think of hackers. Hackers might try to analyse how your application executes and try to understand how your application verifies licences. They will then disassemble it and if they succeed in identifying where¹³ the crucial test is performed, they will change the machine code to bypass that test.

We recommend raising the difficulty for hackers by making licence verification a less deterministic process. Our *Hello World* example is deterministic in the sense that it always checks for licences at the very same place in the code. This makes it easier to identify under what circumstances licences are checked.

As can be seen below, `byssus_check_licensed()` takes two parameters:

```
byssus_error byssus_check_licensed(  
    const unsigned char verification_percentage,  
    const unsigned char label  
);
```

The `label` parameter will be explained later in section 6.1.6. The first parameter is a percentage. In our *Hello World* program, we passed 100 to that function. With this value, licences are validated at every function invocation. However, if we passed 5 instead of 100, the function would actually verify licences in only 5% of function invocations. For the remaining 95% of calls to the function, licences would not be verified and the function would return immediately.

This powerful feature allows `byssus_check_licensed()` to be called more than once and from different places within the application. The percentage parameter should be set to appropriate values so that the net effect is to reliably perform at least one real licence check every time the application is run.

Similarly, if a licence check returns an error, it is a good idea to vary the way the application aborts execution. Implementing licence verification in this manner results in more non-deterministic behaviour making a hacker's task considerably more difficult.

6.1.2 Guidelines For Using `byssus_check_licensed()`

So how do you decide how often and from where in the application `byssus_check_licensed()` should be called?

¹³ Among millions of lines of uncommented assembly language code.

A balance needs to be found between increasing security and keeping the usage of computing resource as low as possible. Start off by analysing the way your application executes in most typical scenarios. How frequently do code paths execute during a typical user session? Which code paths have to execute for users to derive value from the application (e.g. file, print and clipboard operations in a word-processing application)?

The next step is to choose a few locations from where licence verification could be checked.

A call to `byssus_check_licensed()` that does not result in licence verification uses very limited system resources. However, the licence verification process is CPU-intensive¹⁴ so this verification should not be performed more frequently than is really necessary. Placing a call to `byssus_check_licensed()` in a program loop that is executed every few seconds would waste CPU resource. Also note that if the verification frequency parameter is set to a very low value to compensate for the frequency the function is called, then it becomes harder to control the actual number of verifications performed.

Consequently, it is recommended to call `byssus_check_licensed()` from both parts of the application that are executed once only but with certainty (e.g. application initialisation) and from parts of the application that are likely to be executed more than once (but not too frequently). Verification every few minutes would not be a problem. Your function placement should ensure that your application delivers no utility to unauthorised users¹⁵.

Once you have identified the function call locations, then choose values for the verification percentage parameter using your preliminary analysis of application usage patterns to guide you. The aim should be to ensure that licences are physically verified at least once each time the application is run. Although performing additional physical verifications adds no incremental value, the cost of a safety margin that results in a few excess verifications will not be significant if you follow the placement guidelines above.

Note *A successful licence verification result is not stored by the library (for security reasons). Consequently, the physical verification process will continue to occur as controlled by the calls to `byssus_checked_licensed()` and its parameter value.*

API function `byssus_get_stats()` offers help analysing what happens during actual application usage (see section 6.1.6). This function reports the number of times `byssus_check_licensed()` is called, separately for each invocation, and the total number of actual licence verifications performed. Keep tuning the function placement and parameter values until you're happy with the results.

14 Licence keys are encrypted with 2048-bit RSA. Such public key encryption algorithms are compute-intensive.

15 Do not add coding logic specifically to manage the invocation of `byssus_checked_licensed()`.

6.1.3 Probability

Be careful when working out expected verification frequencies and estimating appropriate values for the validation frequency parameter. A sound basic understanding of probability is sufficient, but errors are very easy to make if you don't work carefully and methodically.

Note *The percentage execution functionality is implemented via a random number generator. Consequently, the percentage parameter controls the average frequency of physical licence verification. In practice the actual frequency could vary quite significantly.*

For example, consider an implementation where `byssus_check_licensed()` is called within part of an application that is expected to be called four times per session using the application and that the percentage parameter is set to 25%. On average, physical licence verification would occur once per session from this call location. However, in 32% of sessions physical licence verification would not occur at all¹⁶. If the parameter value is increased to 50%, failure to perform a single licence verification drops significantly to 6% of sessions¹⁷.

Similarly, user behaviour is not uniform. Some users may apply the program in very different ways from others, resulting in very different execution frequencies for different application code modules.

The impact of a failure to verify licences and the consequent possibility that unauthorised usage of an application might take place, depends on the application and how it delivers its value and utility to users. For most applications an occasional failure to verify will still leave the application effectively unusable for end-users because much of the value comes from reliable access to key application features. However, for a few applications, a single unauthorised session might offer significant value and be worth many brute force attempts to get a single uninterrupted session using the application. In this case, you should guarantee that licences are verified by setting the verification frequency to 100% in at least one of the calls to `byssus_check_licensed()`.

6.1.4 Checklist

- Analyse your application code logic and modular structure.
- Analyse how different people use your application.
- Select program locations from which to verify licences
- Choose values for the verification frequency parameter.

16 Probability of no licence validation is $\frac{3}{4}$ per execution of the function. Consequently for four executions the probability is $\left(\frac{3}{4}\right)^4 = 0.32$.

17 $\left(\frac{1}{2}\right)^4 = 0.0625$ or approximately 6%. The expected total number of verifications rises to two per session.

- Test and continue to tune licence verification with the aid of `byssus_get_stats()`, until you get satisfactory results.

6.1.5 Sample – Play The Game

The following is a sample game where we demonstrate non-deterministic licence verification. This sample program asks the user to guess a number between 1 and 100. The verification strategy we use is:

- We include one call to `byssus_check_licensed()` inside the main loop. We are conscious that calling from only one place is not recommended, but as this application is less than one page of code, calling from one place is sufficient here.
- To win, a user needs seven steps with the best strategy, unless he is lucky. If we passed the value that delivers an expected number of verifications of 1, i.e. $100/7$, a third of the games would run with no licence verification¹⁸, which might be too high. Consequently, we pass 20 to `byssus_check_licensed()`. This is a compromise value that on average will cause the licences to be checked more than once per game but ensures that only one in five games run with no licence verification¹⁹.

The code uses the same callback as in *Hello World*. The code itself is easy enough and should be self explanatory:

```
#include <stdio.h>
#include <stdlib.h>
#include <byssus/api.h>

#include "user_interface_integration.h"

int main()
{
#define SECRET_MIN 1
#define SECRET_MAX 100
#define VERIFICATION_FREQUENCY 20

    int secret;
    int nb_steps;
    int guess;
    byssus_error e;

    /******
     * Initialisation phase: application, libraries, etc.
     * *****/

    // Initialise random number generator
    srand((int) & guess); // uses a random int to initialise seed

    // Initialise Byssus API
    e = byssus_init(show_progress, collect_data, show_alert);
    if (e) panic(e);

    /******
     * Execute body of the application
     * *****/
}
```

18 Probability of no licence validation is $\left(\frac{6}{7}\right)^7=0.34$.

19 Assuming 7 steps, the expected frequency of validation is $7 \times \frac{1}{5}=1.4$ and the probability of no licence validation is $\left(\frac{4}{5}\right)^7=0.21$.

```

// Load licences in memory
// (after GUI and other libraries are initialised)
e = byssus_load_licences();
if (e) panic(e);

// Start new game
printf("You must guess a number between %d and %d.\n",
SECRET_MIN, SECRET_MAX);
secret = rand() % SECRET_MAX + SECRET_MIN;
nb_steps = 0;

// Play game
do {
// Input player's choice
printf("Your guess: ");
scanf("%d", & guess);
nb_steps++;

// check licences
e = byssus_check_licensed(VERIFICATION_FREQUENCY, 'A');
if (e) panic(e);

// Show result of player's choice
if (guess != secret) {
printf("too %s\n", guess > secret ? "high" : "low");
}
} while (guess != secret);

// Show score
printf("You won in %d steps\n", nb_steps);

/*****
* Free resources and return to shell
*****/

return 0;
}

```

6.1.6 Statistics

During the test phase of your application development process, you might want to check and tune the rules you chose for using `byssus_check_licensed()`. The Byssus API provides a way to collect basic statistics for the execution of

`byssus_check_licensed()`:

```

void byssus_get_stats(
byssus_statistics * stats);

```

With `byssus_statistics` defined as:

```

typedef struct {
unsigned long nb_requests[UCHAR_MAX];
unsigned long nb_verifications;
} byssus_statistics;

```

So far, the second parameter to `byssus_check_licensed` has not been explained. `label` uniquely identifies the specific invocation location of the function. This allows `byssus_get_stats` to report the number of times `byssus_check_licensed` was called separately, for each invocation location. The labels used could be supplied via an enumerated type defined by the application developer. The label is used as an index in the `nb_requests` array, to retrieve the specific number of calls to `byssus_check_licensed`.

`nb_verifications` reports the total number of real verifications that were performed by the Byssus library.

6.2 The Case Against A Dynamic Library

The Byssus API is delivered to you in the form of a library. As was mentioned in section 2.4, each library Byssus generates contains information identifying that library (and the application with which it is linked) to the Byssus server. Consequently, it is not recommended to share a library between several applications.

It is also very strongly recommended never to convert the static Byssus library into a dynamic library²⁰ (DLL). The reason for never changing the nature of the Byssus library lies in the definition of a DLL: a file containing object code and exposing entry points to external applications.

If the Byssus library were turned into a dynamic library, it would make it very easy for hackers to bypass the licensing mechanisms your application thinks it is using, by creating another DLL emulating the Byssus library entry points and always returning the same value, especially when it comes to `byssus_check_licensed()`.

Reminder *Never turn the Byssus library into a DLL. Always statically link the Byssus library with your application.*

²⁰ Also called a “shared library”.

7 Generation Of Executables

7.1 Compilation Process

All your source files #including <byssus/api.h> must be compiled with the option allowing your compiler to find header files in non standard locations. This option is typically:

```
-I<pathname>
```

The exact parameter provided with this option depends on the location where you installed the Byssus distribution archive in the first place.

You must also compile all files in the “i18n” directory to provide the Byssus library and your application with the right localised messages. See section 5.4 for more information.

7.2 Linking

You need to statically link a total of five files with your application:

- the Byssus library itself, which contains all the code necessary to provide the services described in this document
- all four “i18n” compiled files, mentioned in the previous section.

8 Internal Runtime Operations

This section describes some of the internal operation of the Byssus library, and in particular how and when resources such as main memory, local disk, and network connection are used.

8.1 Licence Verification

Function `byssus_load_licences()` attempts to load licences from the local disk and if unsuccessful, requests temporary licences from the Byssus server. If the trial period has not expired, the server sends temporary licences back which are kept in memory during application execution.

Function `byssus_check_licensed()` tries to match a licence loaded in memory with the combination of the application ID stored inside the code of the Byssus library and with the hardware fingerprints present on the computer.

8.2 Installation And Re-Installation Of Licences

If no licences are present in memory when `byssus_check_licensed()` is called, function `byssus_buy_licences()` is called to manage the payment process. When payment is confirmed, this function receives permanent licences from the server and stores them both in memory and on disk. All this happens automatically.

If an end-user's operating system is re-installed, any licences stored on disk might be lost. No user intervention is required to re-install the licences. `byssus_load_licences()` will request temporary licences from the Byssus server. The Byssus server re-sends the permanent licences when it finds a match against the hardware fingerprints and application identity showing that the licence has already been paid for. The permanent licences are re-stored on local disk. The whole process is automatic.

8.3 Access To The Internet

A network connection to the Internet is needed only when no permanent licences are found on the local disk. This network connection is used either to:

- request temporary licences with `byssus_load_licences()`,
- request licence activation with function `byssus_activate_licences()`,
- process payment with `byssus_buy_licences()`.

8.4 Use Of The Local File System

On Windows, licences are stored in the directory referenced by the default environment variable “\$(ALLUSERSPROFILE)\Byssus” or under “C:\Windows\Byssus” when this environment variable is not available²¹. Under Linux, licences are stored in directory “/opt/byssus/licences”.

²¹ Which is the case with Windows 9X systems.

9 Appendix – Function Reference

All functions provided by the Byssus API are listed alphabetically in the following pages. Each function description includes the following sections:

- Name & functional summary.
- Synopsis: shows the header file needed and presents the function's prototype.
- Description: explains what the function does, how it does it, what parameters it uses, what side effects it might have, and what callbacks it may use.
- Return value(s).
- Restrictions: what functions must be called before, and in what circumstances the function can be used.
- See also: refers to other functions an application may use in conjunction with the one described.

9.1 byssus_activate_licences

Name

`byssus_activate_licences` – Obtain a licence key using an activation code.

Synopsis

```
#include <byssus/api.h>

byssus_error byssus_activate_licences(
    const byssus_form_activation * activation_code
);
```

Description

Activation codes are supplied following proxy purchases i.e. made using a different computer from the computer that will run the licensed application. During licence activation an activation code must be exchanged for a permanent licence key. The activation exchange must be initiated from the computer that will host the application, so that the licence keys match the hardware identifiers.

`byssus_buy_licences` supports interactive licence activation and `byssus_activate_licences` supports automated licence activation. `byssus_activate_licences` should only be used within a purpose-built system management utility dedicated to activating licences for multiple computers. Typically, the utility would be remotely executed under the control of a system management console, on each system needing a permanent licence.

`byssus_activate_licences` takes one parameter representing an activation code, which is sent to the Byssus server. If the activation code hasn't been used before, the server provides a permanent licence key which `byssus_activate_licences` stores on the local disk.

Type `byssus_form_activation` is defined in `<byssus/fields.h>` which is automatically included by `<byssus/api.h>`.

`byssus_activate_licences` does not make use of any callback.

Return value(s)

`byssus_activate_licences` returns `BYSSUS_ERROR_NONE` if successful. If the function returns a different error code, the application must exit.

Restrictions

`byssus_activate_licences` should only be called from system management utilities dedicated to bulk licence activation. It is not part of the payment process, because it does not collect any payment information. `byssus_init` must have been called earlier.

This function requires an active Internet connection.

See also

`byssus_buy_licences`, `byssus_strerror`, `exit(3)`

9.2 byssus_buy_licences

Name

`byssus_buy_licences` – Purchase or activate a licence.

Synopsis

```
#include <byssus/api.h>

byssus_error byssus_buy_licences();
```

Description

`byssus_buy_licences` purchases or activates licences. Internally, this function is called by `byssus_check_licensed` to handle licence payment and activation. However `byssus_buy_licences` should be used directly by the developer to handle most instances of licence activation, and can also be used to offer the opportunity to purchase licences:

- before the expiration of a trial period.
- where trial usage of an application is not made available.

For payment, `byssus_buy_licences` calls the `byssus_input_form` callback function to collect payment information from the end-user. It then contacts the Byssus server to process payment.

When the callback function of type `byssus_input_form` is called by `byssus_buy_licences` with `quantity > 1` or `target_local_computer` equal to `false` then at least one proxy licence purchase occurs. For each proxy purchase an activation code is generated. The activation codes are then sent by email to the email address passed in the `byssus_input_form` callback. If `quantity` is n then n proxy purchases occur if `target_local_computer` is `false`, and $n - 1$ proxy purchases occur if `target_local_computer` is `true`.

If `target_local_computer` is `true`, the server sends permanent licence keys for that computer, which are stored on the computer's disk. These new licence keys are also kept in memory to be validated later by `byssus_check_licensed`.

The licence activation process is very similar to the payment process. The `byssus_input_form` callback collects a licence activation code supplied by the end-user. If the activation code hasn't been used before, the server provides permanent licence keys which are handled exactly as for payment.

`byssus_buy_licences` calls the `byssus_input_form` and `byssus_show_progress` callback functions and might call the `byssus_show_alert` callback function, all previously passed to `byssus_init`.

`byssus_buy_licences` takes no parameter.

Return value(s)

`byssus_buy_licences` returns `BYSSUS_ERROR_NONE` if successful. If the function returns a different error code, the application must exit.

Restrictions

There is no need for the end-user to restart the application after purchasing or activating licences. This function requires an active Internet connection.

See also

`byssus_check_licensed`, `byssus_init`, `byssus_strerror`, `exit(3)`

9.3 byssus_check_licensed

Name

`byssus_check_licensed` – Validates licence keys.

Synopsis

```
#include <byssus/api.h>

byssus_error byssus_check_licensed(
    const unsigned char verification_percentage,
    const unsigned char label
);
```

Description

`byssus_check_licensed` checks whether an application is authorised to execute on the host computer. To achieve this, `byssus_check_licensed` validates the licence keys stored in memory against the Byssus library's identity and the fingerprints of the host computer.

`byssus_check_licensed` allows a developer to randomise where and when unauthorised application usage is blocked, making licence-checking a non-deterministic process. Non-determinism is controlled through the value of `verification_percentage`. Internally, this function generates a random value. If this value is less than or equal to `verification_percentage`, licences are verified. If the random value is greater than `verification_percentage`, `byssus_check_licensed` does not validate the licence keys and returns immediately.

Valid values for `verification_percentage` range from 0 to 100. If 0 is passed, a default value of 15% is used.

For guidelines on how frequently and where to call `byssus_check_licensed` and which parameter values to use, refer to 6.1.

The `label` parameter uniquely identifies each call to `byssus_check_licensed` in the statistics returned by `byssus_get_stats`.

If licence validation is unsuccessful (memory location empty or contains an invalid licence), `byssus_check_licensed` calls `byssus_buy_licences` to present the end-user with an option to purchase or activate a licence.

Return value(s)

`byssus_check_licensed` returns `BYSSUS_ERROR_NONE` if successful. If the function returns a different error code, the application must exit.

Restrictions

`byssus_check_licensed` must be called only after `byssus_load_licences` returns successfully.

`byssus_check_licensed` must always be invoked at least once per execution of the application to make sure the application is protected.

`byssus_check_licensed` does not use `rand(3)` functions. There is no side-effect

for applications using the standard random number generator and using a specific seed to reproduce a sequence of random values.

See also

`byssus_buy_licences`, `byssus_init`, `byssus_load_licences`, `byssus_strerror`,
`exit(3)`, `rand(3)`

9.4 byssus_get_pricelist

Name

`byssus_get_pricelist` – Retrieve the current pricelist for an application.

Synopsis

```
#include <byssus/api.h>

byssus_error byssus_get_pricelist(byssus_pricelist * pricelist);
```

Description

`byssus_get_pricelist` gets the current price list for an application, from the Byssus server. The Byssus server identifies the application by the identifiers embedded in the Byssus library it is linked with.

`byssus_get_pricelist` returns prices in all the currencies supported, in parameter `pricelist` (see 5.5). Each entry in the `pricelist` table is a price value indexed by a value of type `byssus_currency`. The first price in the table is always 0, which corresponds to `BYSSUS_CURRENCY_NONE_SELECTED`.

`byssus_get_pricelist` does not make use of any callback.

Return value(s)

`byssus_get_pricelist` returns `BYSSUS_ERROR_NONE` if successful.

See *Description* above for values returned in parameter `pricelist`.

Restrictions

`byssus_get_pricelist` can only be executed within a callback function of type `byssus_input_form`.

The `pricelist` is provided live from the Byssus server, and is never cached between two executions of an application. This function requires an active Internet connection.

See also

`byssus_init`, `byssus_strerror`

9.5 byssus_get_stats

Name

`byssus_get_stats` – Retrieve statistics on licence verification.

Synopsis

```
#include <byssus/api.h>

void byssus_get_stats(
    byssus_statistics * stats
);
```

Description

`byssus_get_stats` returns statistics about the usage of `byssus_check_licensed` in parameter `stats` which has the following type definition:

```
typedef struct {
    unsigned long nb_requests[UCHAR_MAX];
    unsigned long nb_verifications;
} byssus_statistics;
```

`byssus_check_licensed` is called with a `label` parameter that identifies each specific invocation location. `nb_requests` contains the number of times `byssus_check_licensed` was called from each location in the application, in the array entries referenced by the respective `label` values. `nb_verifications` reports the total number of verifications performed.

`byssus_get_stats` reports statistics from the time `byssus_init` was called.

`byssus_get_stats` does not make use of any callback.

Return value(s)

See *Description* above for values returned in parameter `stats`.

Restrictions

`byssus_get_stats` is a tuning function which can be used during coding and testing but which is not intended to be used in a customer-ready release.

See also

`byssus_init`, `byssus_check_licensed`

9.6 byssus_init

Name

`byssus_init` – Initialise the Byssus library.

Synopsis

```
#include <byssus/api.h>

byssus_error byssus_init(
    const byssus_show_progress progress_fn,
    const byssus_input_form    input_fn,
    const byssus_show_alert    alert_fn
);
```

Description

`byssus_init` initialises all the components inside the Byssus library. This function also performs some checks before enabling the use of other Byssus functions and tables.

`byssus_init` takes three function pointers as parameters. The developer must define these three callback functions, which must conform to the following three prototypes:

```
typedef bool (* byssus_show_progress)(
    byssus_progress_step current_step,
    byssus_progress_step max_step);

typedef bool (* byssus_input_form)(
    byssus_form * data);

typedef void (* byssus_show_alert)(
    const char * message);
```

If the `byssus_show_progress` or `byssus_input_form` callback functions return `false`, then the library will abort whatever operation is currently taking place. Typically the developer will return `false` when an end-user indicates that they want to cancel the current activity e.g. they click on a dialog button provided by the developer for this purpose.

The abort is not guaranteed. The error status reported from the function that calls the callback will indicate whether the abort was actioned or not. For example, once a credit card payment is completed, the payment process can not be aborted. Checks for a specific progress status should not be made (e.g. to attempt to remove an abort dialog button when the overall operation is committed) because the progress status values are subject to change. The end-user should be informed whether the abort request was actioned or not.

`byssus_show_progress` is called by:

- `byssus_buy_licences`
- `byssus_load_licences`

`byssus_input_form` is called by:

- `byssus_buy_licences`

`byssus_show_alert` might be called from:

- `byssus_buy_licences`
- `byssus_load_licences`

Note that all three callback functions could also be called from `byssus_check_licensed`, because `byssus_check_licensed` might call `byssus_buy_licences` internally.

`byssus_init` does not call the functions passed as parameters. It only stores their address inside the library for later use.

Return value(s)

`byssus_init` returns `BYSSUS_ERROR_NONE` if successful. If the function returns a different error code, the application must exit.

Restrictions

This function must be called only once, before any other function in the Byssus library.

See also

`byssus_strerror`, `exit(3)`

9.7 byssus_load_licences

Name

`byssus_load_licences` – Load licence keys into memory.

Synopsis

```
#include <byssus/api.h>

byssus_error byssus_load_licences();
```

Description

`byssus_load_licences` loads licence keys into memory (RAM), making them available to `byssus_check_licensed` (which must be called later).

`byssus_load_licences` operates in three steps:

- It first attempts to load licence keys from the local hard disk.
- If no licences are stored on disk, `byssus_load_licences` contacts the Byssus server to request temporary licence keys. If successful, the licence keys are loaded into memory.
- If the Byssus server identifies that the current application is already licensed for use on the same computer, permanent licences are re-issued to the library. `byssus_load_licences` loads those licences into memory and stores them on the local disk.

During its execution, `byssus_load_licences` may call back a function of type `byssus_show_progress` or `byssus_show_alert` previously passed to `byssus_init`.

`byssus_load_licences` takes no parameter.

Return value(s)

`byssus_load_licences` returns `BYSSUS_ERROR_NONE` if successful. Note that if licences are unavailable because the trial usage period has expired or none is offered, `byssus_load_licences` still returns `BYSSUS_ERROR_NONE`. In this case the licence key memory location will be empty.

If the function returns a different error code, the application must exit.

Restrictions

`byssus_load_licences` should be called only once, preferably before entering the main loop of the application. However, its invocation should be kept as spatially separate from function `byssus_init` as possible. This could be achieved by ensuring that both functions are called from as deep a level of function call nesting as is feasible, and that both functions are called from within different functions in the main program. Placing both function calls one after the other in the main program, should be avoided.

This function requires an active Internet connection.

See also

`byssus_check_licensed`, `byssus_init`, `byssus_strerror`, `exit(3)`

9.8 byssus_strerror

Name

`byssus_strerror` – Get a string describing a Byssus error.

Synopsis

```
#include <byssus/api.h>

const char * byssus_strerror(byssus_error errnum);
```

Description

`byssus_strerror` returns a pointer to a C string describing the error code passed in the argument `errnum`.

That C string is stored in static memory inside the Byssus library.

`byssus_strerror` does not make use of any callback.

Return value(s)

`byssus_strerror` returns the appropriate error description string, or an unknown error message if the error code is unknown.

Restrictions

N/A

See also

`strerror(3)`

9.9 byssus_validate_form

Name

`byssus_validate_form` – Validate the data in the input form.

Synopsis

```
#include <byssus/api.h>

bool byssus_validate_form(
    const byssus_form *   fields,
    byssus_error_report * all_errors
);
```

Description

`byssus_validate_form` validates the data input by the end-user in the `byssus_input_form` callback.

With the exception of `byssus_freeform`, each field is checked for the correct format, invalid characters or out-of-bounds values (syntax checking). Note that payment card checksum digits are validated, to pick up transcription errors during input.

Credit card validity dates are checked against the current date, for logical validity, but no other fields are checked for logical consistency with respect to other fields, nor for factual accuracy. For example, the local sales tax rate is not checked for factual accuracy with reference to both the `country_index` supplied and the trading location of the application vendor. The format of a post code is not checked with reference to the `country_index` supplied, nor is its meaning checked for logical consistency with other address data supplied, such as city or state.

`byssus_validate_form` reports any empty mandatory fields. The application should ensure that all the mandatory fields listed below are clearly identified as mandatory in the form presented to the end-user in the callback function of type `byssus_input_form`:

- `card_number`
- `card_type`
- `country_index`
- `currency`
- `email`
- `expiry_date`
- `holder_name`
- `quantity`
- `security_code`
- `target_local_computer`
- `tax_rate`

`byssus_validate_form` will also require the completion of the following fields if other fields contain specific values:

- `buyer_reference` is mandatory if the optional field, `company_name`, is completed.

- `valid_from` is mandatory when using specific payment cards.
- `issue_number` is mandatory when using specific payment cards.

Parameter type `byssus_form` is defined in `<byssus/fields.h>` and type `byssus_error_report` is defined in `<byssus/errors.h>`. Both files are automatically included by `<byssus/api.h>`.

`byssus_validate_form` does not make use of any callback.

Return value(s)

`byssus_validate_form` returns `true` if all fields are successfully validated. Otherwise it returns `false`, and parameter `all_errors` will contain at least one `error_code` with a value other than `BYSSUS_ERROR_NONE`.

Restrictions

`byssus_validate_form` must be used inside a callback of type `byssus_input_form`, which in turn is called by `byssus_buy_licences`. When `byssus_input_form` returns to `byssus_buy_licences`, `byssus_buy_licences` calls `byssus_validate_form` as a catch-all safety mechanism. If `byssus_validate_form` returns an error, `byssus_buy_licences` will call the `byssus_input_form` callback again, and so on until the internal call to `byssus_validate_form` returns successfully. Under these circumstances the end-user would probably be presented with a blank form to re-fill after each failed internal validation check made from `byssus_buy_licences`. This must be avoided. Consequently, `byssus_validate_form` should be regarded as a mandatory function and should be used carefully.

`byssus_validate_form` performs no check on the custom field of type `byssus_freeform`.

See also

`byssus_init`, `byssus_buy_licenses`, `byssus_strerror`