PRODUCT MANUAL

Dynamic C®

Integrated C Development System
For Rabbit Microprocessors

iDigi Services for Rabbit Developer's Guide

90001161_A

iDigi Services for Rabbit Developer's Guide

Part Number 90001161_A • Printed in U.S.A.

Trademarks

The latest revision of this manual is available at www.rabbit.com.

TABLE OF CONTENTS

1. Introduction

The iDigi for Rabbit API is a set of libraries for use with a Dynamic C programming environment. It allows any Rabbit 4000, 5000 or 6000-based device with an Ethernet or WiFi communications port, and at least 512kB of RAM, to connect to an iDigi server.

iDigi solves some difficult problems for deployment of devices with a communications interface, and allows the application developer to concentrate on their field of interest without having to worry about network management, data storage and remote firmware updates.

The Rabbit iDigi API presents a simple interface for performing the following tasks:

- Robust network configuration management, with backup settings
- Logging data to a central (iDigi) server
- Remote firmware update
- Customizable remote command and query execution.

All of this is available if there is an available connection to the Internet, at least part of the time but preferably with permanent access. It works behind firewalls, or with direct Internet connections.

Security is provided via SSL (or TLS).

It is easy to update an existing application (written with the Rabbit TCP/IP API) to take advantage of iDigi. Basically, most of the application which is concerned with managing the network configuration can be removed. The program main loop, which normally calls `tcp_tick()`, is modified to call `idigi_tick()` instead.

Existing libraries which have a role in network and firmware management, namely ADDP and the Remote Program Update facility, are automatically integrated when iDigi is used. There is no requirement for additional programming.

1.1 Sample Programs

Use of iDigi in an application is simple enough that it is most easily described using sample programs. Samples are in samples\iDigi under the Dynamic C installation folder. The sample programs are listed below.

IDIGI_SIMPLE_TLS.C

This is the first sample to run. It provides a “bare-bones” iDigi sample which enables remote network configuration of the target board, with the connection secured via TLS. This and all other iDigi samples have comments at the top of the program which provide instructions for setting up an iDigi test account, and compiling the program.

At its simplest, iDigi provides the ability to reconfigure network settings from the iDigi server, via a web-based application. Since this application can run on any PC, anywhere, it is possible to reconfigure devices anywhere in the world, without fear that an inadvertent misconfiguration will render the device inaccessible.

IDIGI_UPD_FIRMWARE.C

This sample should be run after the one above, if only to ensure that a successful connection to the iDigi server can be obtained. This sample shows how to configure an application so that it supports remote firmware update via the iDigi server.

Programmatically, firmware updates are enabled simply by defining the `IDIGI_USE_RPU` preprocessor macro. This sample has fairly complex instructions, however end-users need not be concerned with this if they are provided with a binary firmware image file.

IDIGI_PUT_DATA.C

This shows how to upload data from the application to the iDigi server. The library handles most of the details of using the iDigi Data Services. The application decides where the data will be placed on the server account (folder and file name) and provides the data content via a memory buffer.

IDIGI_DO_COMMAND.C

This demonstrates customization of a device to support a “do command”. Digi devices such as the X2 which act as gateway devices support the Python programming language for customization. The Rabbit does not include a Python interpreter, however it is easy to register C functions to perform whatever custom processing is required in an iDigi application. Rather than allowing upload of new Python code to modify functionality, the Rabbit assumes that most custom functions are not intended to be modifiable by end-users other than by completely replacing the firmware.

2. iDigi.lib API Configuration Macros

This section defines the iDigi API configuration macros, functions and data structures which are documented for use by developers.

2.1 Feature Selection

The following macros specify inclusion of various subsystems. Their use can cause a lot of extra code to be included, and hence use a lot of flash memory.

#define IDIGI_USE_TLS

If defined, include SSL/TLS code for secure connections

#define IDIGI_USE_ADDP

If defined, include ADDP for convenient device provisioning. Note that an ADDP callback function is automatically installed which is compatible with iDigi. Only the `ADDP_PASSWORD` macro needs to be defined.

#define IDIGI_USE_DS

If defined, include code to use iDigi Data Services (i.e. ability to store data on the iDigi server using HTTP or HTTPS PUT). Since this facility uses the HTTP client library, it sets the HTTP client mode using `httpc_set_mode(HTTPC_NONBLOCKING | HTTPC_AUTO_REDIRECT)`. The application should not change the non-blocking setting.

#define IDIGI_USE_RPU

If defined, include Remote Program Update library to allow firmware updates from the iDigi server. In this case, you will also need to define the following macros in the project settings “defines” box:

```
_FIRMWARE_NAME_="MyFirmware"
_FIRMWARE_VERSION_=0x0101
```

The firmware name is not directly significant to iDigi, however it is used as a default for `IDIGI_PRODUCT` (see below). It is recommended to define `_FIRMWARE_NAME_` to be the same string as `IDIGI_PRODUCT`, provided the name is 19 characters or less.

NOTE: The length of the `_FIRMWARE_NAME_` must not exceed 19 characters otherwise firmware updates will fail.

The `_FIRMWARE_VERSION_` should be two hex bytes (combined into 16 bits as shown). The version number should increment for each firmware release, in order to be able easily to identify the firmware ver-

sion which is running and connected to the iDigi server. In the iDigi user interface, this version number appears in the “Firmware Level” column, as a dotted decimal with two leading zeros e.g. 0.0.1.1.

2.2 Device Personality Selection

The following macros define the device “personality”. The defined values represent the defaults. Your application can override these by defining the macros before `#use idigi.lib`.

It is highly recommended to define at least `IDIGI_PRODUCT` to a non-default value. `IDIGI_PRODUCT` and `_FIRMWARE_VERSION_` together define a unique “key” for the iDigi server. The server caches certain (static) information about each device+firmware, keyed by this combination of values.

```
#define IDIGI_PRODUCT _FIRMWARE_NAME_

#define IDIGI_VENDOR “Unknown”

#define IDIGI_VENDOR_ID “0”

#define IDIGI_FIRMWARE_ID “1.00.00”
```

The default for IDIGI_PRODUCT is the value defined for `_FIRMWARE_NAME_`. It is recommended to keep these macros the same. iDigi uses `IDIGI_PRODUCT` not `_FIRMWARE_NAME_`, however the latter macro is significant to the Remote Program Update facility, which is used by iDigi to manage firmware updates. `IDIGI_FIRMWARE_ID` is an arbitrary string, and is only significant in that it appears in the iDigi user interface under the System Information/device_info/Firmware version field. To avoid end-user confusion, this string should be made the same (or similar to) the `_FIRMWARE_VERSION_` macro, rendered as dotted decimal.

The following are used as initial values. They can be set using iDigi web services or user interface.

```
#define IDIGI_CONTACT “None”

#define IDIGI_LOCATION “Unknown”

#define IDIGI_DESCRIPTION “Unknown”
```

2.3 Macros Which Must Be Defined

These macros must be defined since they have no defaults. The defined values are provided as examples only.

#define IDIGI_SERVER “sd1-na.idigi.com”

The iDigi server fully qualified domain name. It may be the name of a global string variable. This is an initial default. The iDigi server can set this to a different value. The macro `IDIGI_SERVER_CURRENT` returns the current server FQDN string.

#define IDIGI_SERVER_PASSWORD “idigi”

The password for accessing the server. If never defined, the password is an empty (zero length) string. This is only used as an initial default. The password (_RCI_Settings.devicesecurity.password) is usually set via the iDigi server.

#define IDIGI_USERBLOCK_OFFSET 0

A byte offset into the userID block at which to save the iDigi network configuration and other settings. If this is not defined, a warning is issued, since the ability to remember settings over a reboot is important to most applications. Currently, the application programmer needs to manage the offsets and sizes of objects in the userID block. If the application makes no other use of the userID block, then it is advantageous to allow iDigi to access the entire block (set this macro to 0 and the following macro to the entire userID block size i.e. SysIDBlock.userBlockSize)

#define IDIGI_USERBLOCK_MAX_LEN 8192

Maximum amount of userID block storage dedicated to saving the iDigi network and other settings. This is used the first time the settings are saved. Thereafter, the length is saved in the storage area itself, and will not be decreased, however it will be increased if this macro value is increased.

2.4 Other Macros

#define IDIGI_MIN_CONNECT_TIMEOUT 30

Define to specify an initial connection timeout (sec), from the point at which the network is brought up and an iDigi server connection is attempted, to the establishment of an open connection. Default 30 seconds. If a backup configuration is available (i.e. a connection was established using a previous configuration) then that backup configuration will be attempted.

#define IDIGI_MAX_CONNECT_TIMEOUT 300

Define to specify the maximum desired timeout for iDigi server connection. This should be reasonably long to prevent unnecessary flipping between network configurations in the case that e.g. a cable is disconnected. The initial connection timeout (given by `IDIGI_MIN_CONNECT_TIMEOUT`) is applied to the first connection attempt (e.g. after a reboot). This macro determines the timeout value for subsequent attempts.

#define IDIGI_DEBUG

If defined, turns on debugging for all iDigi subsystems.

#define IDIGI_VERBOSE

If defined, turns on debugging printfs for all iDigi subsystems. This can cause shortage of root constant space. If so, add ROOT_SIZE_4K=9 in the project defines box, and turn on the separate I and D option in the compiler settings.

#define IDIGI_IFACE_VERBOSE

If defined, causes messages related to the network interfaces and iDigi connection to be printed. This causes a lot less output than IDIGI_VERBOSE, yet it is still useful for debugging connections.

2.5 Using Remote Program Update with iDigi Applications

RPU is a Rabbit facility which pre-dates the availability of iDigi. The firmware update facility in iDigi uses the RPU library to actually perform the firmware update in an efficient and robust manner.

RPU is enabled in an iDigi application by #define IDIGI_USE_RPU. When done, this automatically causes inclusion of board_update.lib. *Rabbit Application Note 421* describes use of this library in detail. The following macros are significant to RPU, and are also used by iDigi in order to promote a consistent view of the target device.

```
#define _FIRMWARE_NAME_ "GyroSensor Mk II"
#define _FIRMWARE_VERSION_ 0x0101
```

The _FIRMWARE_NAME_ definition is propagated to iDigi as the default value for IDIGI_PRODUCT. It is possible to define IDIGI_PRODUCT independently, however it is recommended to keep the same definition since the macros basically define the same firmware identification string.

_FIRMWARE_VERSION_ is an arbitrary 16-bit value, which feeds into a 32-bit identifier used by iDigi. The larger sized identifier exists because iDigi supports non-Rabbit devices which have larger identifiers. On the Rabbit, since RPU existed first and used 16 bit numeric identifiers, this value is simply re-used in place of the larger identifier by padding it on the left with zeros. Thus, the identifier as seen by the iDigi server is 0x00000101, which is typically rendered as "0.0.1.1".

There is currently no requirement to change the version number from the default 0x0101, since it is used by the iDigi server to determine whether the firmware has been upgraded in a manner which is significant to the iDigi user interface. Since iDigi itself is not customizable on the Rabbit, the firmware version number does not need to be changed. Note that customization of an iDigi application via registration of different "do commands" is not considered to be a "significant" firmware change by the iDigi server, and thus changes to the registered commands do not need to have an incremented firmware version number.

There is, however, a good reason to increment the _FIRMWARE_VERSION_ number for each new public release of a product firmware. That reason is that the current firmware version appears in the iDigi user interface devices table (under the "firmware level" column). When upgrading firmware, it is useful to provide feedback that the new firmware version is correctly installed, thus it is recommended to increment the number for each release.

There is a related versioning macro, IDIGI_FIRMWARE_ID, which is basically historical and does not have any significance other than appearing in the iDigi user interface under the System Information panels. It is recommended to make this appear similar to the _FIRMWARE_VERSION_ macro value as it appears

in the devices list, in order to avoid user confusion, and thus should be a dotted decimal whose last two fields equal the _FIRMWARE_VERSION_ value e.g. “X.1.1”.

When RPU is enabled in an iDigi application, the procedure for updating the firmware is basically as follows:

1. The programmer creates a new .bin file as normal for a Rabbit Application. This is described in detail in the IDIGI_UPD_FIRMWARE.C sample program. End users do not need to be concerned with this process; they only need the resulting binary firmware image. Note that the file extension (.bin) should not be changed, otherwise the target device will reject the firmware update. In an effort to avoid simple mistakes, the target rejects any file which does not have a .bin extension. The rest of the name is not significant to the target.
2. In the iDigi user interface, the end-user selects (highlights) the target device(s) which are to be updated, and executes the “update firmware” menu option.
3. The user enters the firmware .bin file name in the dialog box, and hits the “update firmware” button. This will transmit the new firmware to all selected target devices, and reboot them.
4. In the device list, the new firmware version number should be displayed. If any device experiences a failure, the previous firmware version will be displayed. The RPU library ensures that updates are robust. If the new firmware is fully tested (and does not have any crash bugs) then the update will either succeed, or the old version will continue to run.

2.6 Code and Data Memory Requirements for iDigi Applications

The iDigi library makes extensive use of dynamically allocated memory, and thus does not significantly impact on root data memory use. All Rabbit-provided library code uses the system memory space, `_sys_malloc()`, leaving use of `malloc()` memory entirely to the user application.

The following table shows memory usage for the `IDIGI_DO_COMMAND.C` sample program. Numbers indicate memory usage in kilobytes. For the RCM5450W, no encryption was used except: WPA - Wifi Protected Access (Personal) encryption added; EAP - WPA Enterprise authentication added.

Table 1-1.

Core Module	Options				RAM					SYS malloc memory	
	ADDP	RPU	TLS	DS	RootCode	XMEMCode	RootConst	RootVar	XMEMVar	HWM	Idle
RCM4200	No	No	No	No	26k	245k	1k0	5k	21k	45k	28k
	Yes	No	No	No	27k	253k	11k	5k	21k	45k	28k
	Yes	Yes	No	No	28k	263k	11k	5k	26k	45k	28k
	Yes	Yes	Yes	No	29k	335k	12k	9k	31k	64k	48k
RCM4550W	No	No	No	No	28k	288k	11k	6k	59k	45k	28k
	Yes	Yes	No	No	28k	307k	12k	7k	67k	45k	28k
	Yes	Yes	No	Yes	28k	32k0	12k	7k	64k	45k	28k
	Yes	Yes	Yes	No	29k	380k	13k	1k	68k	64k	48k
RCM5450W +WPA	Yes	Yes	Yes	No	3k0	408k	14k	12k	70k	64k	50k
RCM5450W +EAP	Yes	Yes	Yes	No	30k	501k	17k	10k	70k	64k	50k

From the above table, an estimate of the additional resources for each iDigi-related feature may be obtained:

Table 1-2.

Option	Additional Memory					
	RootCode	XmemCode	RootConst	RootVar	XMEMVar	SYS malloc
Addp	1k	8k	1k			
RPU	1k	10k			5k	
DS		13k		1k		
TLS	1k	73k	1k	4k	5k	19k

If the application is failing to compile because the compiler cannot fit the application in the available memory, then the following methods may help resolve the problem.

1. For any iDigi application, use a core module with at least 512k program space and 256k data memory. If using EAP on a WiFi module, at least 1M code space is required. Although it is possible to use WPA on a module with 512k code space, it restricts the size of the application code, thus a 1M module is recommended for any iDigi application which also requires any form of WPA.
2. Adjust one or more of the following parameters in the compiler options settings:
   - Turn on "separate instruction and data" if not already checked. In rare cases, better memory utilization can be obtained by turning separate I&D off. This is only the case if the total root code, constants and variables add up to less than about 48k, which may allow slightly more memory to be used by xmem code, variables, and dynamic allocation.
   - In the "defines" tab, add definitions for the following macros:

     ROOT_SIZE_4K = 7

     This specifies a basic dividing line for root memory allocation. The lower part is specified by this number (multiplied by 4096), with the upper part specified by the remaining memory up to about 47k. Memory from 47k up to 64k is devoted to special use like the stack and xmem code window.

     With separate I&D on, `ROOT_SIZE_4K` specifies the available space for root constants (especially C string literals) at the expense of root variables, with root code able to take up the full 47k. With separate I&D off, then this number indicates the total amount for root code plus root constants, at the expense of root variables.

     XMEMCODE_SIZE = 0x70000

     This increases the amount of space that can be used for code. On a module with 512k of code space, the maximum value for this number is about 0x78000, since 32k is normally reserved for the Rabbit system- and user-ID blocks which are used to store configuration and calibration data.

     Adjust the values shown above and recompile
3. Once the application can be successfully compiled and run on the target board, if the application runs out of dynamic memory (system malloc) then:
   - Add the macro `_MALLOC_HWM_STATS` to the defines tab. This enables simple statistics for dynamic memory usage.
   - Modify the application so that at the start of `main()`, the following code is inserted:

     ```
     _init_sys_mem_space();
     _sys_malloc_stats();
     ```

     This initializes the system memory space, and prints its size and usage (usage should be zero at this point since no allocation has been performed). You can insert additional calls to `_sys_malloc_stats()` at various points in the code (e.g. after calling `idigi_init()`) in order to detect where memory exhaustion is occurring.

     It is also possible to add a detection of keypress (`kbhit()` and `getchar()`) in the main loop which calls `idigi_tick()`, and print the memory statistics at that point.

- The following macro can be set in the defines tab:

  _SYS_MALLOC_BLOCKS=32

  Adjust the value (which is in units of 4k memory blocks) to avoid dynamic system memory space exhaustion. The maximum allowable value depends on the amount of free RAM on the board, and how much is required by the application memory space (ordinary `malloc()`). The default is 16 (giving 64k system RAM), however for applications which require TLS this should be changed to 32 (for 128k). You can insert the following code at the start of `main()` to print a listing of the available xalloc memory areas, from which the system memory space is obtained:

  xalloc_stats(xubreak);

  "xubreak" is an internal library global variable which is the start of a BIOS-generated list of available xmem blocks. The system memory is typically allocated from the largest of these blocks. System dynamic memory cannot be split over two such blocks.

3. Developing iDigi Applications on the Rabbit

3.1 Feature Selection

The first step in application development should be to decide which features are required. The requirement for robust security, #define IDIGI_USE_TLS, adds the largest overhead. IDIGI_USE_DS (data services) adds an HTTP client. Use of RPU and ADDP add a relatively small overhead. See section 2.6 for memory requirements.

If an existing application is being upgraded to use iDigi, then the iDigi library will probably be able to make use of much of the existing library code. In particular, the following existing subsystems will be reused by iDigi:

- TLS/SSL (if secure HTTP or WPA enterprise in use)
- Dynamic memory allocation (malloc.lib)
- General ethernet or Wifi networking (dcrtcp.lib)
- RabbitWeb
- HTTP client (for data services)
- Remote Program Update (RPU)
- Advanced Device Discovery protocol (ADDP)

If iDigi is used, it may be possible to remove existing network configuration code since this is completely handled by iDigi. An exception may be if a local configurator is required, which cannot be replaced with ADDP, such as a serial port terminal.

The following table shows the amount of additional code and data memory when adding basic iDigi support to existing applications. The first column of the table lists a standard Rabbit sample program, and the other columns show the additional memory used when the sample is modified to #use "idigi.lib" and the main loop is changed to call `idigi_tick()`.

Table 1-3.

Sample Program Dynamic	RootCode + XmemCode(*)	RootConst	RootVar	XMEMVar	System Malloc
pong.c	+226k	+9k	+4k	+59k	+45k
tcpip\http\static.c	+126k	+8k	+2k	+2k	+40k
tcpip\rabbitweb\humidity.c	+81k	+7k	+2k	+2k	+40k

(*) In general, iDigi does not have any root code requirement. The sum of root and xmem code size may be used when estimating additional code memory requirements.

pong.c is a program with no existing network functionality, thus addition of iDigi to this program adds the most amount of code. The other samples selected have more existing functionality which is re-used by the iDigi code, thus less additional memory is required for these samples.

3.2 Converting an existing application

The following steps outline the procedure for converting an existing application (assumed to contain some networking code such as an HTTP server) to an iDigi-enabled application.

a) Replace the following code (usually at the top of the main C code):

```
#define TCPCONFIG 1
#use "dcrtcp.lib"
```

with this:

```
#define IDIGI_PRODUCT "<my product>"
#define IDIGI_VENDOR "<my company>"
#define IDIGI_VENDOR_ID "<my vendor ID>"
#define IDIGI_FIRMWARE_ID "<my firmware id>"
#define IDIGI_CONTACT "<contact email>"
#define IDIGI_LOCATION "<location>"
#define IDIGI_DESCRIPTION "<description>"
#define IDIGI_SERVER "<initial iDigi URL>"
#use "idigi.lib"
```

Add macros to select options if desired (IDIGI_USE_ADDP etc.).

b) In the main initialization code, replace

```
sock_init_or_exit(1);
```

with

```
if (idigi_init())
    exit(1);
```

c) In the main application loop, at least insert a call to

```
idigi_tick();
```

This can replace a call to `tcp_tick()`, however it does not replace a call to `http_handler()` or any other specific network protocol handlers.

Robust applications should use the style of main loop shown in all the iDigi samples. In particular, check for the return code from `idigi_tick()` and perform the appropriate actions.

3.3 Non-volatile storage

When incorporating iDigi into an application, or writing a new application, the developer needs to keep in mind the fact that iDigi stores settings in non-volatile memory. Without iDigi, the initial state of a program is determined entirely by compile-time defaults, such as provided by configuration macros. When iDigi is used and connectes to an iDigi server, then network settings may be changed and stored in non-volatile memory. It is the new settings which may be used the next time the program is run (even if re-compiled and reloaded). This can be surprising to developers who are used to Rabbit programming.

Normally, this behavior should not be too troubling, however if the developer is experimenting with various iDigi features, then sometimes the configuration saved in non-volatile memory can conflict with options selected in the new program. this is particularly so in the case that a subsystem has been omitted. For example, if you initially run with TLS enabled, but subsequently re-compile without TLS, then the saved settings may be telling iDigi to use a secure connection when the required code is not even included. This will cause permanent errors at start-up.

One way of getting around this problem is to start from scratch each time. If the macro _IDIGI_FORCE_FACTORY is defined, then this will bypass the initial read of the previously saved settings. It may be handy to always define this macro during development, but remember to remove it prior to deployment.

3.4 Initial Configuration

The most likely problem when initially developing an iDigi application is that the board will fail to make an initial connection to the iDigi server. The initial connection is critical, and depends on compile-time defaults. If the defaults are invalid for the local network, then the connection will never be established. The problem of initial configuration when new devices are deployed must be solved by local configuration means.

If the intended network environment is guaranteed to have a DHCP server, then that is the most convenient means of initially provisioning devices without end-user involvement. It is also wise to include ADDP, since that will allow local configuration (via a laptop or PC) even if there is no DHCP server.

The other critical piece of information is the URL of the iDigi server. An initial default for this is selected via the IDIGI_SERVER macro. See “Macros Which Must Be Defined” on page 9.

4. API Functions

Rabbit's iDigi implementation is comprised of the following API functions:

idigi_init()	initialize all of iDigi and start/maintain network interfaces.
idigi_tick()	non-blocking driver for all iDigi functionality
idigi_status()	query current iDigi connectivity status
idigi_secure()	test if iDigi connection secured by TLS/SSL
idigi_register_target()	register a custom `do_command` target function
idigi_put()	use iDigi data services to save data on the iDigi server
idigi_ds_tick()	non-blocking processing of a PUT operation
web_error()	used to indicate errors to the server for custom targets.

idigi_init()

SYNTAX

```
int idigi_init(void)
```

DESCRIPTION

Initialize iDigi and the network.

Non-iDigi applications call `sock_init()` or related functions to start up the network. When using iDigi, all network configuration is handled automatically and the application should *not* call `sock_init()`.

Registration of `do_command` targets should not be performed until this function has been called.

RETURN VALUE

`0: OK`

`-ENOMEM: insufficient memory`

`Any other: internal error, contact technical support`

LIBRARY

`idigi.lib`

idigi_tick()

SYNTAX

```
int idigi_tick(void)
```

DESCRIPTION

Drive all state machines for maintaining iDigi and the network configuration. Your main application loop should call this function whenever possible. Before calling, `idigi_init()` must have been called successfully.

RETURN VALUE

Integer code as follows:

0	OK, keep calling
-NETERR_NONE	A remote configuration change has been received which requires one or more network interfaces to be temporarily shut down. This code can be ignored (treat like 0) or the application can cleanly shut down any open connections before calling `idigi_tick()` again. This return code can be ignored if iDigi is the only network connection used in the application, or if only standard library servers (such as HTTP or FTP) are in use. If client sessions are in use e.g. the application connects to a database server and wishes to shut down cleanly, then the application should perform the necessary cleanup (including calls to `tcp_tick()`) before resuming calls to `idigi_tick()`.
-NETERR_ABORT	A reboot request has been received. Application should perform any clean-up, then reboot using exit(0) from main(). Alternatively, this request can be ignored, however this may cause a surprise to web services clients or iDigi user interface users. Note that a reboot is requested after firmware updates.
Other	Generally, these will be negative network error codes. They can occur if the network is misconfigured and the iDigi server cannot be reached. If ignored, `idigi_tick()` will try again in 3 seconds.

LIBRARY

`idigi.lib`

idigi_status()

SYNTAX

```
int idigi_status(void)
```

DESCRIPTION

Return iDigi server connectivity status.

RETURN VALUE

Integer code as follows:

IDIGI_DOWN	Not connected or invalid state.
IDIGI_COMING_UP	Attempting to connect
IDIGI_UP	Connected OK
IDIGI_COMING_DOWN	Temporarily bringing network down for reconfiguration.

LIBRARY

idigi.lib

idigi_secure()

SYNTAX

```
int idigi_secure(void)
```

DESCRIPTION

Return TRUE if iDigi server connectivity is secured via TLS/SSL.

RETURN VALUE

0 if not secure

1 if secured via TLS

LIBRARY

idigi.lib

idigi_register_target()

SYNTAX

int idigi_register_target(char far * name, char far * request, char far * reply)

DESCRIPTION

Register an iDigi do_command target.

EXAMPLE

```
struct {
       int a;
       char b[20];
} request;
#web request
// example validity checker:
#web request ($request.a != 13 || \
                                  web_error("13 is unlucky for 'a'!"))


struct {
       float b;
       int z[4];
} reply;
#web reply


void actionRequest(void);
#web_update request actionRequest
void actionRequest(void) {
       reply.b = request.a + 3.14159;
       ...
}


int main() {
       idigi_init();
       idigi_register_target("myTarget", "request", "reply");
       ...
}
```

NOTE: All parameters must point to static storage, since only the pointers are stored in the registered target table.

PARAMETER 1

The name of the target, as it would appear in the target attribute of the <do_command> element. If a target of this name is already registered, its entry is updated with the following parameters. Otherwise, it is created.

PARAMETER 2

Name of the variable which is filled in by the request parameters. This variable must be a structure, unless the command has no relevant parameter data, in which case it should be the name of a simple int variable. The variable must be registered to RabbitWeb via a #web directive.

All targets must have a request parameter. Your application defines a callback function which is invoked for each do_command received for that target. Use #web_update to register the callback function against the relevant request parameter variable.

If multiple targets use the same request parameter variable, then the update function should update all possible reply variables, since it won't know which target was actually specified by the server.

PARAMETER 3

Name of the variable which is used to generate reply data. This may be NULL if there is no reply data. Otherwise, it must be the name of a structure variable (not a simple int etc.!) which has been registered in its entirety using #web. This variable is used to define the structure of any data returned in the reply. As such, it need not have any guard or update callbacks of its own. It will usually be manipulated directly by the update function of the request variable in order to generate the reply.

RETURN VALUE

Integer code as follows:

0	OK
-ENOMEM	More than IDIGI_MAX_TARGETS registered. IDIGI_MAX_TARGETS defaults to 10, but you can #define it to a larger value before #use idigi.lib.

LIBRARY

idigi.lib

idigi_put()

SYNTAX

```
int idigi_put(DataSvcsState_t far * dss, char far * name, int secure,
   char far * contenttype, void far * data, word len);
```

DESCRIPTION

Use iDigi data services to put file (or folder) to the server.

This function is only available if you `#define IDIGI_USE_DS`.

Only one `PUT` operation may be in progress at the same time! Attempting multiple operations will result in application crash.

NOTE: Some of the required parameters for communicating with the server are taken from the current RCI state. These parameters are:

`_RCI_Settings.mgmtglobal.dataServiceEnabled`

If not set TRUE, then this function will fail with return code `-EPERM`. Defaults to TRUE.

`_RCI_Settings.mgmtglobal.dataServicePort`

Default 80. Sever port number for plaintext requests.

`_RCI_Settings.mgmtglobal.dataServiceSecurePort`

Default 443. Sever port number for secure requests.

`_RCI_Settings.mgmtglobal.dataServiceURL`

Defaults to "`/ws/device`", and is a prefix to the "name" parameter.

`_RCI_Settings.mgmtglobal.dataServiceToken`

Defaults to "`cwm_ds`"

PARAMETER 1

Pointer to uninitialized state structure. This will be initialized by this routine, then it must be passed to `idigi_ds_tick()` until it returns something other than `-EAGAIN`.

PARAMETER 2

Resource name to create, relative to the device-specific root. For example, "`foo.xml`" to create a file /foo.xml, or "`bar/baz.txt`" to create a file baz.txt in folder /bar. The create a folder, pass NULL for data.The length limit for the resouce name is 128 characters.

This name should be URL encoded (e.g. spaces should be %20 and so on).

PARAMETER 3

TRUE if secure connection to be used, else will use plaintext connection. This can only be set TRUE if `IDIGI_USE_TLS` is defined.

PARAMETER 4

String to send as "Content-Type". Use NULL for default of "text/plain". This string must not be changed until `idigi_ds_tick()` completes.

NOTE: Currently, the iDigi server ignores the specified content type, and infers the content type from the filename extension. It is recommended to pass NULL for this parameter.

PARAMETER 5

Data to put. This data must remain unmodified in-place until `idigi_ds_tick()` completes. If NULL pointer is passed, this means create a folder. Otherwise, a file is created (or replaced) on the server.

PARAMETER 6

Length of data to put (typically strlen(data)). A maximum of 65535 bytes is supported.

RETURN VALUE

Integer code as follows:

`0`	Success. Call `idigi_ds_tick()` until it completes.
`-ENOMEM`	Could not allocate local resources
`-EPERM`	Not permitted because `_RCI_Settings.mgmtglobal.dataServiceEnabled` is not set TRUE.
`-EACCES`	No access because 'secure' parameter TRUE but no TLS connection is available.
`Other`	Any return code from `httpc_put_ext()`.

NOTE: See also "*iDigi Web Services and Programming Guide*" located on the Documents page after logging into www. iDigi.com.

LIBRARY

`idigi.lib`

idigi_ds_tick()

SYNTAX

```
int idigi_ds_tick(DataSvcsState_t far * dss);
```

DESCRIPTION

Use iDigi data services to put file (or folder) to the server. This function is only available if you `#define IDIGI_USE_DS`.

Call `idigi_put()` first to initialize the state structure. This function is used to continue and complete the process, which may take a relatively long time.

Always call this function with the same `DataSvcsState_t` structure, until it returns something other than `-EAGAIN`. If you do not complete the process in this manner, then there may be a resource leak.

PARAMETER 1

Pointer to state structure initialized by `idigi_put()`.

RETURN VALUE

Integer code as follows:

positive value `PUT` completed, with this code returned by the server. Typically, on success, this will be:

- **201** Resource created

May also get the following error codes:

- **400** Bad request
- **403** Access forbidden (bad credentials)
- **503** Service unavailable

-EAGAIN Not complete, call again with unchanged data.

-EINVAL Bad parameter: dss appears not to be initialized correctly.

Other Any return code from `httpc_put_ext()`.

NOTE: For details on the positive return codes, see RFC2616.

LIBRARY

`idigi.lib`

web_error()

SYNTAX

```
int web_error(char far *error)
```

DESCRIPTION:

This function may be invoked from #web variable guard expressions, to generate informative error messages.

EXAMPLE

```
int myvar;
#web myvar ($myvar < 16 ? 1 : web_error("Too big"))
```

This works because the return value of web_error() is always zero, and hence causes the correct result for the guard expression in the case that there is an error.

An alternative style of usage is:

```
#web myvar ($myvar < 16 || web_error("Too big"))
```

which takes advantage of C short-cut evaluation to produce the same result as the first form.

NOTE: This function must only be called from #web guard expressions, since it depends on some global information which is set up during #web transaction processing.

PARAMETER 1:

Pointer to a null-terminated string containing the error message. When used with iDigi do_command targets, this string appears in the <hint> element of the <error> XML element. Any error generated by a guard expression (whether or not accompanied by web_error()) causes the entire do_command to be rejected i.e. there will be no update action.

RETURN VALUE

0

NOTE: See also "*iDigi Web Services and Programming Guide*" located on the Documents page after logging into www. iDigi.com. SCI is used by client programs to post do_command XML requests to one or more iDigi-connected devices, including those based on Rabbit modules.

LIBRARY

rweb_generic.lib