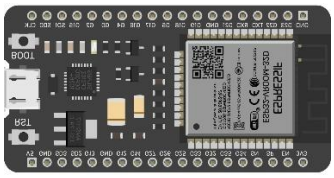
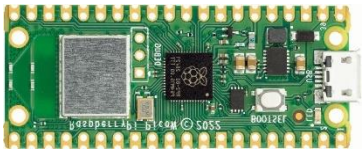
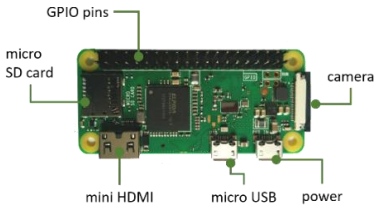
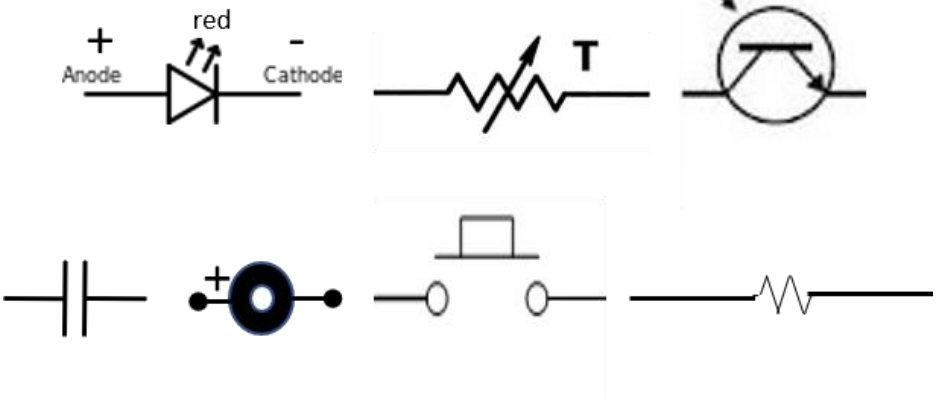
IoT Tiki usage





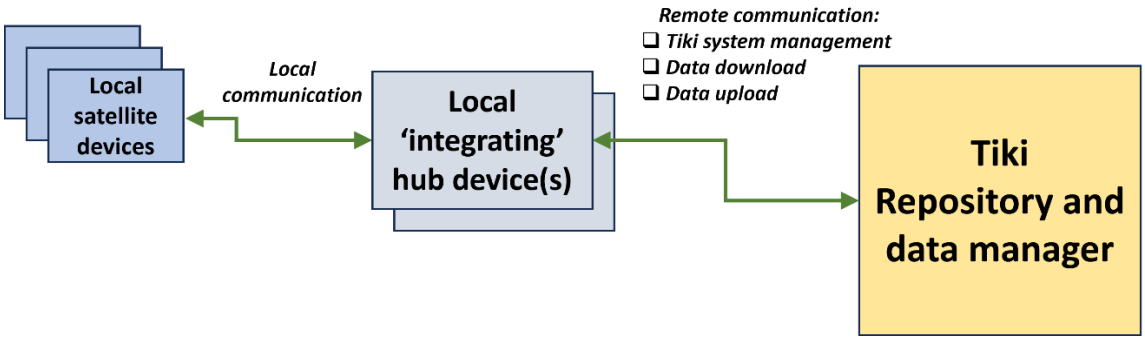
version 1.0

Table of contents

Introduction

This evolving document aims to provide some notes on how Tiki can play an important role in the deployment and use of the "Internet of Things" (IoT) which is very multi-faceted and involves an ever expanding range of products and services.

The schematic below provides a very generic picture of how the various components of an overall 'Tiki IoT' system might be deployed.



- **Local satellite devices**: multiple low-cost single board computers (SBCs), simpler microcontroller modules or dedicated sensor devices e.g., with built-in RF comms – all typically, with low power consumption and/or 'deep sleep' modes to facilitate long operational duration when just battery powered.
- **Local communication**: communication from a satellite device to a local hub is usually a wireless method and could be 433MHz RF including LoRa RF, WiFi, or just Bluetooth for close proximity. Directly wired connections from a satellite to a hub are of course possible and could use any of the usual 'bus' protocol or an ethernet connection could be used.
- **Local 'integrating' hub device(s)**: there would usually be just one hub device at a location, but multiple hub devices may be needed, and they are all usually low-cost single board computers (SBCs), but in some exceptional circumstances they could be simpler microcontroller modules. These hub devices gather data from defined sets of multiple satellite devices - although sensors could also be directly attached to the hub device which directly manages the sensor and the data collection – to distinguish this from a satellite being directly connected which simply passes a 'result' to the hub.
- **Remote communication**: from a local integration hub, that via the internet, provides access to a central Tiki instance. This communication is enabled with 'programming' on the local integration hub device to control secure access, with the use of an access 'token,' to the Tiki API that allows various simplified and fast data upload/download options.
- **Centralized Tiki**: Tiki supports many business-process operational methods, often using specialized plugins. A central Tiki system, accessible in a secure manner by multiple distributed administrators and other users, with a hierarchy of access permissions, can support IoT system and remote device management, as well as collected data storage, plus management reporting, and analysis that can initiate Tiki user notifications when specific events are 'seen'.

Each of these areas are now described in some more detail.

IoT system deployment components

Local satellite devices

An IoT system does not have to deploy multiple 'satellite' devices - but many do, mainly when each satellite device is a very simple, often single function, sensor.

Home automation systems are a good example of a satellite arrangement, and for example ***Energenie*** supply a range of satellite devices for controlling electrical sockets, detecting door/window opening/closing, PIRs for general movement detection, etc. Usefully they also provide a Raspberry Pi SBC add-on board (***ENER314-RT***) that is both a 433MHz transmitter and a receiver, supports both OOK and FSK RF transmission methods, and also allows data encryption for added security. Core software for the ENER314-RT board is provided by Energenie but more developed code is available from ***https://github.com/whaleygeek/pyenergenie***.

Energenie also use a well-developed connected devices messaging structure called OpenThings that was developed by Sentec in 2014 - but this specification no longer seems to be maintained or supported by Sentec (or anyone else?), although an archived copy of the original description document can be downloaded ***from here***.

Most satellite devices will be a fully integrated device i.e., incorporating a sensor and a local communication capability – all typically controlled by a microcontroller which could be a very basic PIC device or a more complex device like the ESP32 or RP2040. Given the widely available and broad range of sensors and very versatile microcontrollers that support several types of communication method, almost any custom satellite sensing device can be developed with currently available technology.

Local communication

Local communication from satellite devices to a local hub device tends to always use some form of wireless transmission, although directly 'wired' options, using one of the many 'bus' protocols, could be used when a satellite sensor is in close proximity to the hub device.

The decision regarding a communication method for satellite devices that are more remote from the local hub comes down to the useful 'range' of a wireless method and the complexity of the 'handshakes' that are needed to make the transmission reliable and secure.

433MHz RF transmission is a very common wireless method with a number of options that theoretically can provide a wireless range measured in km's (!) but the range does depend upon the RF antenna used and the transmission power provided (and allowed by regulation!). Very basic antennae and low power may only allow about a 100m range, which is good enough for (say) home automation. However, changing the protocol can also help, e.g., LoRa (it simply stands for Long Range!) is a spread spectrum method that uses short data messages and quite modest power, but it's protocol makes it relatively immune to signal degradation and interference and can apparently be used over distances of more than 1km if there is good line-of-sight transmission.

WiFi can also be used as a local communication method, with all the satellites, as well as the hub, connected to a local WiFi network. The hub device can then be configured as a server that 'knows' all the satellite local IP addresses and 'waits' for 'requests' that have

data set out in a defined structure. This WiFi network approach is useful as it can 'break up' the range issue with an intermediate set of WiFi access points that can each be wired to a local network or wirelessly meshed together.

Bluetooth is another wireless option, that theoretically has a range of up to 100m for a Class 1 device, or as low as 1m for a Class 3 device (although Bluetooth 5.3 is now supposed to support up to 240m of range?)

Local 'integrating' hub devices

The logic for deploying a local 'integrating' hub device is to allow the very detailed aspects of individual sensor usage and initial data collection to be carried out on dedicated satellite devices, each focused upon their own specific measuring/monitoring activity - but which can each be programmed to report back to the integration point in a consistent way. That integration point will also be 'local' so that communication from satellites can use whatever local communication method makes most sense from a proximity, complexity, and security point of view.

For example, the Energenie home automation system's range of 'satellite' devices can all report into a Raspberry Pi using 433MHz RF when it is fitted with the ENER314-RT. They also use a standard data structure for the RF communication with the Raspberry Pi managing all the satellite device communications, carrying out basic data integrity checks, storing the data locally, as well as being able to do data uploads to a central Tiki.

As already mentioned, in some cases where a single, isolated sensing location is involved, there may not be a need for any satellite devices and instead a number of sensors could be directly connected to a local hub device.

Example software and documentation for a very wide range of sensors and other devices that connect to a Raspberry Pi SBC or various types of microcontroller can be downloaded from ***here*** with general descriptive information provided ***here***.

Remote communication

Communication from a local integration hub to a central Tiki will nearly always use WiFi from the local hub to a local network WiFi access point, although a fixed wire ethernet connection to a switch/router on the local network can obviously always also be used. The local hub can then connect to the internet to access the domain where the central Tiki system is running i.e., just like a browser does, but just for a set of specific transactions that can be actioned programmatically by the local hub device and interpreted by the Tiki API.

Transactions for IoT usage from a Tiki API usage perspective should include at least the following:

- Uploading/downloading of files to/from Tiki File galleries.
- Addition of new, updating existing or downloading existing Tiki tracker items.
- Looking for specific content on a designated wiki page or in a tracker item.

This document, in later sections, discusses an approach to support Tiki API access by a hub device with the development of a series of standardized software functions that could

be run on small single board computer (SBC), like the range of Raspberry Pi SBCs, as well as some of the common microcontrollers.

Central Tiki support for such communication is already substantially provided by the Tiki API, but further programmable access to the Tiki API by a local hub device has been needed for IoT usage and is documented here.

It should be noted that onward support for any local hub developed software is not considered part of the usual Tiki system/software support. Instead, providing background and context support through full documentation of the Tiki API and making available working examples of developed local device access software functions, will go a long way to successfully enable IoT usage with Tiki.

Centralized Tiki

Tiki, as well as supporting an API as discussed above, also provides a powerful content management and business-process platform. This enables its use as a central IoT repository and overall system manager since it can be easily ‘configured’ in various ways to store, test and present data, generate structured reports, and automate actions such as generating emails when defined situations occur.

Much documentation already exists on Tiki and its various features and usage, so with the exception of discussion of the Tiki API, no further details about Tiki itself are provided in this document.

Some IoT usage scenarios using Tiki

Set out below are a number of targeted usage scenarios that are envisaged for an overall Tiki IoT system deployment with the components described above.

These are purely descriptions about the aims of each usage scenario and the general processes involved.

Later sections of this document provide more technical details about the Tiki API functionality to be used, the detailed software needed for an 'integrating' hub device to connect to the Tiki API and the communication return responses that are relevant to these scenarios.

The targeted key uses are as follows:

(1) Collecting sensor data and storing it in a Tiki tracker

This first IoT scenario is possibly the most useful that will arise for the following reasons:

- Data, once collected from a local sensor, and as appropriate forwarded to a local 'integrating' hub device, will be managed and controlled by that local hub device. The data will typically be stored locally, however being able to immediately 'post' the data to a remote repository provides additional data security, i.e. by using Tiki as an immediate and up-to-date off-site storage capability.

- Also, using a Tiki tracker allows data to be stored in a structured manner (that can be easily 'evolved'!) facilitating routine analysis and review. Graphical presentation of the data will, however, be an important analysis technique that is required within Tiki.
- The data can also be continually reviewed and analysed by different/remote groups of users with access control to the data controlled down to an individual field level.
- Finally, sensor data can come in a variety of formats (text, date/time, numeric, logical, a file, etc.) - all of which can be accommodated in a specific tracker field type.

So typically, an 'integrating' hub device like a Raspberry Pi or similar single board computer (SBC), will manage a network of local sensors collecting data on a periodic basis or triggered by a specific event such as a sensor detecting movement, or a temperature threshold being exceeded. The hub device then needs to upload each set of new data, as quickly as possible to Tiki.

This most common usage scenario therefore demands a number of particular requirements for the data transfer and how it is managed by Tiki;

- As new sets of data can arrive very frequently, the Tiki upload process needs to be fast and the Tiki date/time resolution/display needs to be at the level of seconds
- For good general management and data validation/tracking purposes, the sensor data storage tracker on the Tiki system should be configured to allow the following to be recorded:
  - the time/date of the data set upload;
  - the Tiki userId for which the API access token is authorised;
  - the details of the specific 'integrating' hub device doing the data set upload;
  - the details/versions of all the software doing the upload from the 'integrating' hub device; and
  - any relevant operational details/updates/status of the hardware and attached sensors used in the IoT network, e.g. the % full of any local storage media.

(2) Accumulating other types of information and storing it in or retrieving data from a Tiki File gallery

For several usage scenarios, the ability to archive, retrieve, or just make available remotely, a simple file rather than an explicit structured data set, may make the use of a Tiki File gallery preferable to the use of Tiki trackers. It should be noted however that since a Tiki tracker also has a 'Files' field, a tracker could still be used to add structured data to an individual file upload!

Some example usage situations are as follows:

- Images captured locally either on a periodic basis, e.g., perhaps for time lapse video creation purposes, or on an event basis, e.g., triggered by movement detection in a security/surveillance system context.
- Routine upload of various types of 'log' file that are tracking the performance and/or 'health' of local sensors or other hardware; this could either be needed on a periodic basis or on an event basis, e.g., the upload of a system 'log' file, along with the generation of an email notification, if a local storage medium (USB stick etc.) reaches a designated capacity used level.

(3) Using a control file for 'integrating' hub device operational management

To make the onward control of a local 'integrating' hub device as simple as possible, a number of key software system parameters may be defined in a control file where the parameter values may need to be varied depending on the deployment situation or periodic environment changes.

This control file could of course be updated by direct local access to the local hub device, but this is not always possible e.g., if the site is remote. Also, in many situations to avoid security issues, direct access to a local hub device either from a local intranet or direct from the internet may need to be avoided i.e., the hub device should only be allowed to 'talk out' to the internet.

The hub device should therefore be programmed to retrieve a control file from Tiki on a periodic basis but only if that central Tiki file is 'newer' than the locally held version.

An XML-like file structure provides a flexible approach for such a control file and a customised Tiki Plugin has already been developed (***XMLUpdate***) that allows such a file, stored in a Tiki File gallery, to be updated from a Tiki wiki page. In addition, since updates to a Tiki File gallery file are time/date stamped, an existing plugin (***Files***) can be used to display the file's latest modification date in a formatted wiki page, and this timestamp can be programmatically checked by the local hub device to 'see' if a new control file should be downloaded if its local copy timestamp is also tracked and stored locally on the hub device.

(4) Remotely initiating an operation on an 'integrating' hub device

In the context, as described above, where direct access to a local hub device either from a local intranet or direct from the internet may need to be avoided - a safe and secure method of executing hub device system level operations on an occasional, or indeed exceptional, basis is needed.

This could be accomplished by a specialised routine running periodically on the hub device that checks the value of a designated tracker item, that is only accessible by designated Tiki 'device users', and where the tracker item value defines an "operation instruction" with some specific text, or some private code, or is just a reference number.

In this way a remote user with the appropriate permission to access (and edit) the tracker item can 'use' any one of a potential and confidential 'list' of possible operation codes. Each 'instruction' once correctly identified by a hub device could just initiate a one-line system command or a more complex series of commands. Some examples might be:

- do a simple system reboot;
- upload to Tiki the latest version of a particular system log;
- backup/archive a local storage medium, e.g. a USB stick, to an off-site storage facility; or
- a particularly complex operation would be to update a local software program to a new version, which would typically need a series of steps that would be specific to that software program, such as:
  - copying the current software code to a temporary location, so that a roll-back process could be done if necessary;

- downloading the new software, to another temporary location;
- during a designated time-window, overwrite the existing software with the new version or if using a new file name place the new software in the 'production' software folder and as appropriate update any process, e.g., crontab, used to run the program;
- on completion of all the key steps, do a remote edit of the Tiki tracker item to change the 'operation instruction' to text that 'says' the instruction has been completed, so that the next periodic check cycle does not repeat the command;
- as necessary, reboot the local hub device system.

For all such scenarios the code development and all its complexity are primarily centred upon the 'integrating' hub device and this is the responsibility of the hub device developer. The Tiki API communication related code just has to:

1. 'detect' a recognised instruction from a tracker item; and to
2. update the tracker item to 'report' operation completion or perhaps some other status.

Local 'integrating' hub software

The role of a local 'integrating' hub device is to:

- gather data from defined sets of multiple satellite devices - although sensors could also be directly attached to the hub device which directly manages these sensors as well as the overall data collection; and to
- manage the connection to the central Tiki system using the Tiki API.

The software used on the hub device is not part of the Tiki system and will often be custom code, although there are many proprietary and open source systems 'out there', particularly serving the home automation requirement – a good example is the open source Domoticz system (https://domoticz.com).

A main purpose of this report is to document the development of software functions that can be used to connect to the Tiki API and can be easily integrated with whatever systems/software are being used on the local 'integrating' hub device.

Choice of language and method for the development of Tiki API communication functions is considered critical and the current approach reported here is to use 'C' with cURL as the primary candidates since:

- standard 'C' can be readily integrated with Python and is also natively supported on a wide range of SBCs and microcontroller platforms; plus
- cURL and the libcurl library provide a comprehensive range of networking functionality.

'C' code libcurl function development

In order to prototype some working functions for a local integration hub to access the Tiki API using cURL, some 'C' functions have been developed using the ***libcurl library*** and Raspberry Pi's as the development hardware platform.

The aim has been that, by using standard 'C', this should be able to provide a common code base that can be deployed, with almost none or only very minor changes, to a variety of local hub hardware platforms. This code, and any variants that arise, can then be publicised/promoted as working examples for Tiki API access from devices with embedded software, to support wider use of Tiki as an IoT platform.

The initial development was carried out on a Raspberry Pi4 with the Bullseye OS, with later development on a Raspberry Pi5 and the Bookworm OS, and latterly a CM4 with Bookworm has been used for additional testing.

For Python integration, 'gcc' was used to compile a *control_iot_YYMMDD.c* file which has a number of functions, along with its associated *control_iot_YYMMDD.h* file, to create a shared library *libcontrol_iot_YYMMDD.so* - where *YYMMDD* in all these file names is a date code (year, month, day) used to provide version control for the files as they are expected to evolve over time. At the time of writing this report *YMMDD* was *240403*.

The *libcurl* library was installed on the various Pi platforms as follows:

```
sudo apt-get install libcurl4-openssl-dev
```

The gcc compiler is part of the Pi's standard OS and the compilation command used on the Raspberry Pi (with its specific file paths) to create the *.so* file was as follows:

```
gcc -shared -o /your_file_path/libcontrol_iot_YYMMDD.so -fPIC
/your_file_path/control_iot_YYMMDD.c -I/usr/local/include -L/usr/local/lib -lcurl
```

The *control_iot_YYMMDD.c* file should be considered as reasonable working code, but should be thoroughly tested for any production usage, and Appendix A provides current descriptions of all the individual functions that have been developed.

Local 'integrating' hub device Python code development

Python test programs have also been written, using the Python ***ctypes library***, to call the individual 'C' functions.

The *ctypes* library is included by default in Python so no extra installation should be needed.

Appendix B provides current descriptions of the Python test programs, which should also be considered as reasonable working code.

Local 'integrating' hub device 100% 'C' code development

To further extend the development, the same set of 'C' functions were then 'used' by an example 'C' main routine (`IoT_Ctest_240403.c`) replacing the Python test programs. The 100% 'C' code should then be more portable, and it has already worked OK on the Raspberry Pi5 by using the following gcc compilation command to produce an *.exe* file which was then run successfully.

```
gcc -o /your_path_to_compiled_result/Ctest_IoT_240403.exe
/your_path_to_the_main_file/IoT_test_240403.c /your_path_to/control_iot_240403.c -
I/usr/local/include -L/usr/local/lib -lcurl
```

For reference Appendix C provides further description of the main *IoT_test_YYMMDD.c*, *program* which should just be considered as a demonstration of what is possible.

Local 'integrating' hub platform testing

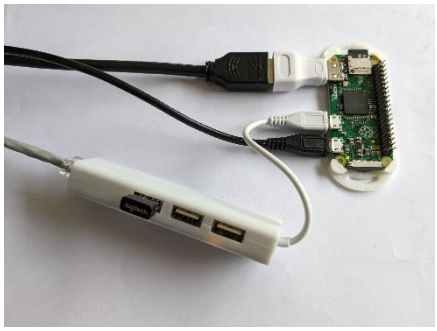
The 'standard' set of C code has been tested on a number of different platforms as discussed below:

Raspberry Pi 4

The initial development and first set of tests were carried out from Aug'23 onwards on a Raspberry Pi 4 with 8GB of RAM that currently runs the 32bit Bullseye OS. This device was being used as a general purpose test system and was always connected to a custom PCB with multiple sensors and other devices that are all 'attached' to the Pi's GPIO pins. This therefore provided a readily available 'integrating' hub device that could be connected to a Tiki instance.

The early development and testing, through to October'23, demonstrated the basic functionality of some initial 'C' code that allowed sensor data to be automatically uploaded to a tracker running on a Tiki24 instance. It, importantly, also identified the 'gaps' and issues that were apparent in the Tiki24 API, and this early work initiated the setting up by **AvanTech** of a new, dedicated IoT testing Tiki instance. This new test platform could be periodically updated to the latest version of Tiki trunk in order to finalise and test a 'good enough' set of IoT 'C' capabilities in time for the scheduled Tiki27 release.

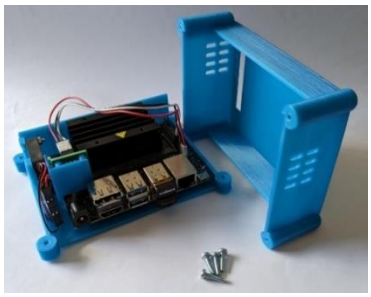
Raspberry Pi Zero



To test the 'C' code on a smaller/less powerful Raspberry Pi SBC, a very old Pi Zero v1.3 was set up from scratch with a 'fresh' install of the latest Pi variant of Debian 11/Bullseye (the Pi version of Bookworm had not been released at the time) – and the only other configuration was to install the libcurl library as described earlier. This Pi Zero only had 512MB of RAM and didn't even have WiFi, so as shown in the image on the right, an adaptor was used to provide a wired ethernet connection.

The same set of 'C' code files as were used on the Pi4 were recompiled on the Pi Zero and the resultant tests ran exactly as expected.

Jetson Nano Development Kit



The Nvidia Jetson Nano provides an example at the other end of the 'power' range of small SBCs, and the Development Kit made available by Nvidia packages the small Jetson module on a carrier board that exposes nearly all the available interfaces and also fits a substantial heatsink on top of the module. The images on the right show the Nvidia Development Kit further packaged in a custom 3D printed enclosure with a cooling fan.

The same set of 'C' code files, as were used on the Pi4, were also recompiled on the Jetson Nano and the resultant .exe file as well as the Python test code ran exactly as expected.

Raspberry Pi 5

A Raspberry Pi 5 was then used as the main hub device test vehicle. The Pi5 was only announced in October ’23, and it is a considerable upgrade on the previous ‘top of the line’ Raspberry Pi 4.

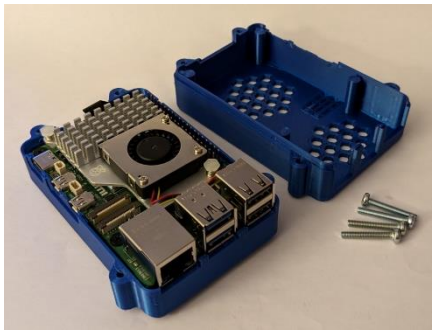
The unit used, shown on the right, had 8GB of RAM and had been fitted with the standard ‘Active Cooler’ accessory, a combined heatsink and cooling fan.

Also, as the layout of the ethernet and USB ports has been changed from that used in the Pi4, a new 3D printed case design was used - also shown on the right.

As expected, the same set of ‘C’ code files, as were used on the Pi4, were also recompiled on the Pi5 and the resultant .exe file as well as the Python test code ran exactly as expected.

Another new feature of the Raspberry Pi 5 is the availability of a single lane PCI Express (PCIe) connector (details **here**) which has allowed a 500GB NVMe SSD to be added to the Pi5 using the Pimoroni **NVMe Base**.

The Pi5 set up was further developed to boot from the SSD, so this ‘all electronic’ type of assembly with no moving parts provides a very robust IoT platform. The images below show the updated assembly in a new custom 3D printed case developed for the assembly.



Raspberry Pi Compute Module 4

Additional testing and a time lapse project were carried out during February and early March ’24 on a CM4 with 2GB of RAM and 32GB of eMMC, fitted to the CM4 IO Board, and running the Bookworm 64bit OS. This assembly was housed in a Waveshare enclosure with a cooling fan and an external antenna as shown in the images below:

Following further API updates and ‘C’ code refinements, all the CM4 testing ran as expected.

Tiki configuration for IoT usage

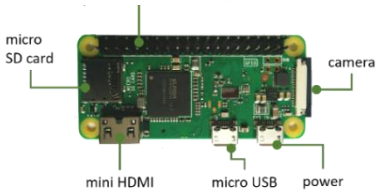
Any Tiki deployment used as an IoT repository and system manager will need to be configured in many different ways, but the specifics to support the 'transactions' to and from a local integration hub device should include the following configurations by the Tiki admin:

- Create a new user Group (with its own home/landing page is usually a good addition), with appropriate global permissions, for all the 'users' (devices) that will have the same access to a set of Tiki resources (wiki pages, File galleries, trackers, etc). Please note this may be just one user used for a 'collective' set of local 'integrating' hub devices.
- Create a new Category that assigns whatever access control permissions are appropriate to the new Group above.
- Create at least one new user (i.e. a device) assigned to the Group above – perhaps using a userId equivalent to the device name, or the 'collection' of hub devices.
- Activate API access and create access tokens (tick box and create button at the 'API' tab in the 'Security' Control Panel). Create tokens associated with each user (device) that has been set up.
- Set up appropriate Tiki resources, categorized appropriately, for use by individual users (devices), e.g., File galleries; Trackers; Wiki pages; etc.
- A final small change is to also update the 'Short time format' in the "General" Control Panel, 'Date and Time' tab – to something like %H:%M:%S %Z, i.e., to include seconds and the time zone designator (the default usually just shows hours and minutes). This will then allow a tracker 'Date and Time' field, which may be used in a sensor data upload, to discriminate and display down to seconds, i.e., to cover the situation where data could be uploaded multiple times within a single minute duration, and to explicitly show the time zone.
- Related to Tiki time/date formatting it should be noted that when 'posting' data to a "Date and Time" tracker field from an integration hub device, this should always use an integer representing the Linux epoch time for the event and not a text string representing the date and time, as any other method risks creating lots of errors within Tiki which needs to store date/time data as an epoch integer so that it is completely unambiguous and various standard Tiki configurations are then used to set the time zone and display formats.

Hardware platforms for local 'integrating' hub devices

The following sections provide further details on how various hardware platforms can be used to access the Tiki API using the prototype 'C' code.

Raspberry Pi single board computers (SBC's)



The recently released Raspberry Pi 5 shown on the left, is a major update to the Raspberry Pi 4 shown below right.

Both models are part of the family of Raspberry Pi SBC's. Testing has so far been carried out on both a Pi5 and a Pi4, each with 8GB of RAM as well as an old Pi Zero v1.3 (shown on the left).

But all the Pi's in the SBC family can use the same standard operating system (a Debian variant), although the 64bit version of Bookworm cannot run on some of the very early/smaller Pi models. The Pi5 tests were carried out with the Pi running the latest Bookworm 32bit OS, but most developed Python and 'C' code should run on any of the machines.

Successful hub device 'C' code usage has already been demonstrated with these three devices where the Pi4 and Pi5 are quite highly spec'd and the old Pi Zero version is a much lower spec', i.e., it only has 512MB of RAM.

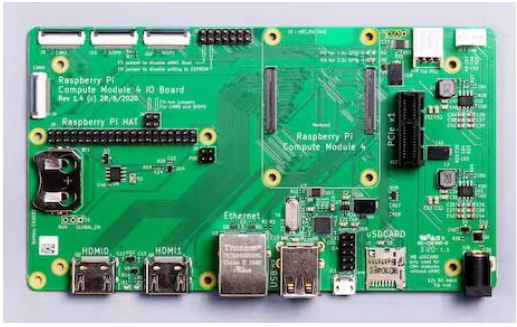
However, of additional interest for IoT local 'integrating' hub deployments are the 'Compute Module' variants within the Pi SBC family (shown on the right and see ***here*** for more details).

The 55x40 mm Compute Module 4 (CM4) provides the power of a Raspberry Pi4 in a compact form factor especially developed for embedded industrial applications and comes in 32 variants of RAM, eMMC Flash, and with/without wireless connectivity.

The current rumour is that its successor, the CM5, providing the power of a Raspberry Pi5, will arrive sometime in 2024.

The CM4 is designed to connect to a system board, that would be specifically developed for the industrial application, via two 100-pin high density connectors.

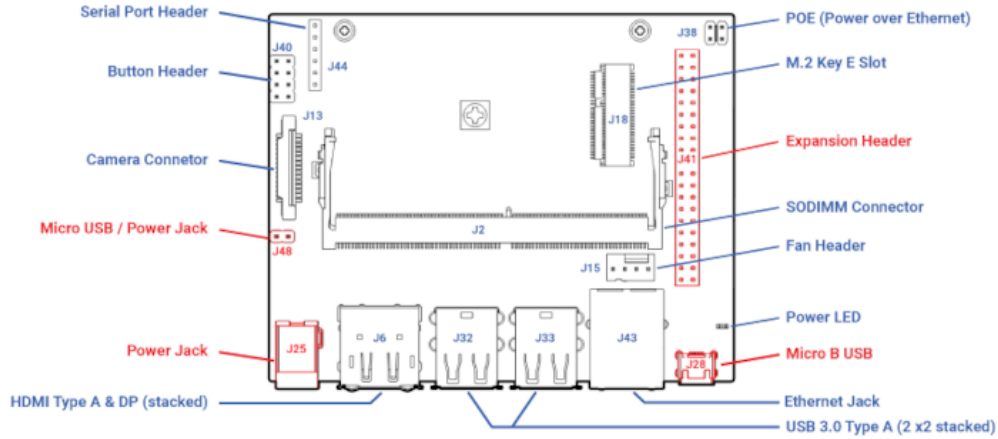
However, to assist the development of products that use the CM4, a CM4 IO Board powered by either 5VDC or 12VDC, and shown on the right, is made available (more detail ***here***) that exposes all the IO normally available from a Pi 4 as well as a PCIe socket, a RTC with a battery socket that can 'wake up' the CM4, two MIPI CSI-2 camera FPC connectors, and two MIPI DSI display FPC connectors.

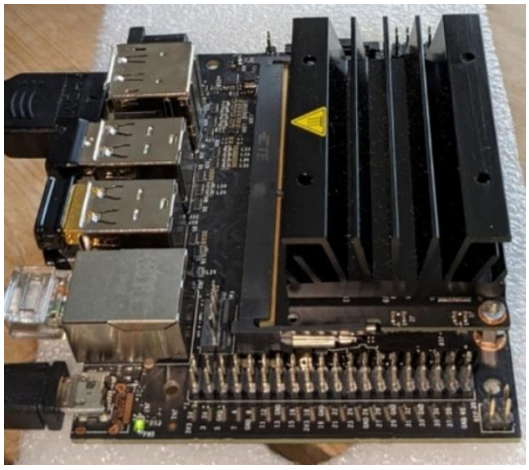


Jetson Nano development kit

The Jetson Nano module, shown on the right, is a 75x45mm production-ready System on Module (SOM) that is targeted at embedded industrial AI applications that would be part of a custom developed overall system.

Nvidia do however offer a Development Kit which has a custom carrier board, schematic shown below, and a large heatsink (!), that provides convenient exposure of most of the system interfaces and where the 2x20 GPIO expansion header also mimics the standard Raspberry Pi layout!

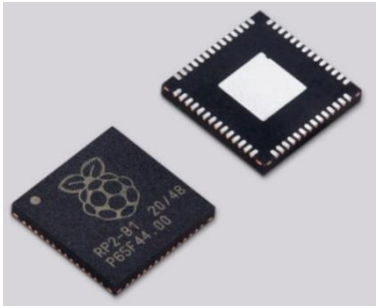




The Development Kit comes fully assembled and the image on the left shows the unit that was used for testing, the 945-13450-0000-000 version that was made available in early 2019 – a newer version is now available.

RP2040 microcontroller

The RP2040 microcontroller chip, shown on the right, was designed by the Raspberry Pi Foundation and first launched in January 2021 powering the Raspberry Pi Pico and other RP2040-based products from other producers. The chip on its own was then made available to anyone in single-unit quantities in June 2021.



The Raspberry Pi Pico module has 264 KB of RAM, 2 MB of flash memory and is programmable in C, C++, Assembly, MicroPython, CircuitPython and Rust.

The Pico W, shown on the left, was introduced in June 2022, initially supporting only 802.11n WiFi, but with the launch of version 1.5.1 of the 'C' SDK in June 2023, the Bluetooth capability on the Infineon CYW43439 used on the module was enabled.

The Pico can be programmed using the Arduino IDE but use of the official SDK (***details here***) is recommended.

The Raspberry Pi Pico is a probable good choice as a very low cost local hub device – but as yet the 'C' code for Tiki API communication has not been adapted for this platform.

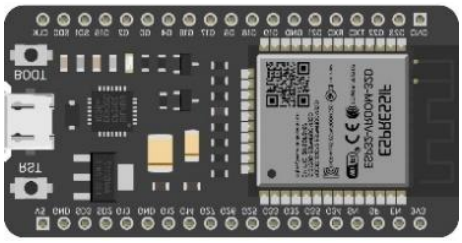
This is therefore a further useful future development.

ESP8266 microcontroller

The ESP8266, although only a modestly powered microcontroller, is also a probable good choice as a very low cost local hub device – but as yet the 'C' code for Tiki API communication has also not been adapted for this platform.

This is therefore also a further useful future development.

ESP32 microcontroller



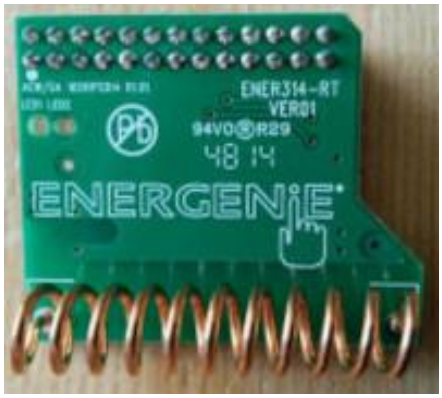
Finally, the ESP32 is a very capable microcontroller and is therefore also a probable good choice as a very low cost local hub device – but as yet the 'C' code for Tiki API communication has also not been adapted for this platform.

This is therefore also a further useful future development.

Some IoT deployment projects

The following section provides short outlines of some active projects in order to illustrate typical IoT use cases.

Home Automation



This project uses an Energenie ENER314-RT 433MHz RF transceiver, shown on the right, connected to a Raspberry Pi 4 acting as a local integration hub. The Raspberry Pi and its add-on board are housed in a custom 3D printed case as shown in the two images below.

The custom system code, written mainly in Python, uses compiled 'C' to manage the 433 MHz RF signal handling as well as 'IoT data' uploading to a central Tiki repository.

The system uses Energenie 'satellite' devices that switch power, detect movement from PIR sensors and door opening/closing from magnetic sensors, as well as overall power consumption within the house from a CT (current transformer) based sensor.

The 'integrating' local hub system software:

- sends RF signals to switch lights on/off at designated times;
- receives the house power consumption at periodic intervals; and
- receives movement and door opening/closing messages on an event basis.

Movement readings from any of the five deployed PIR sensors and the single door open/close sensor are uploaded to a tracker on a central Tiki repository as the events happen, and house power consumption is uploaded periodically to another tracker.

Data is also stored locally on a USB drive connected to the Raspberry Pi as well as being stored on a local NAS.

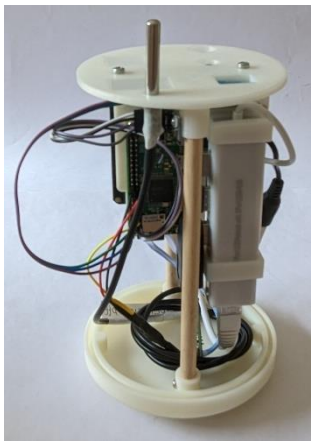
Air Quality Monitoring system

This project demonstrates the use of a Nova SDS011 sensor that uses a laser scattering technique to measure particle concentrations in the air from 0.3 through to 10 µg/m3 (*see **this link** for more detail on various techniques for particle size analysis*). Data output from the SDS011 sensor provides direct readings of PM2.5 and PM10 (where PM stands for Particulate Matter) that are the two readings most used to assess air quality.

The SDS011 uses a fan, that draws in 'fresh' air that is 'measured' and then expels the air back out again, and the SDS011 has been installed in a custom overall assembly.

The internal component assembly, shown in the first two images below, is the un-sleeved unit shown upside-down - where custom 3D printed fixings have been used to create a complete system of:

- the SDS011 sensor
- a DHT11 humidity and temperature sensor,
- a DS18B20 temperature sensor,
- a BMP180 pressure and temperature sensor
- a Raspberry Pi Zero, used to control and manage the system, with
- a 3 USB port + ethernet port adaptor to expand the Pi's connectivity options.




The final sleeved assembly with two wall fixing 'rings', still shown upside-down, is shown in the 3rd image above and the 4th image shows the unit positioned outdoors on a wall (the right way up!) in its final production deployment position.

The measurement regime, controlled by the Pi, collects data every 30 minutes where the temperature, humidity and pressure measurements are single readings but the PM2.5 and PM10 values are averages of 50 readings taken with a short interval between each reading - where the full measurement cycle of all the readings and their storage typically takes about 85 seconds.

The unit's connectivity and power are provided through a single ethernet cable (about 6 to 7m in length) where passive splitting of power and data are carried out using an adaptor within the overall assembly and at the far end of the ethernet cable which connects to the local network and a 5VDC power supply.

The collected data is stored locally on a USB drive as well as being uploaded to an online Tiki tracker using compiled 'C' routines accessed from the overall Python programming.

Appendix A: *control_iot_YYMMDD.c* description

This .c code, and its associated .h file, providing a set of Tiki access functions, whilst it has undergone significant testing should be considered as early development 'quality' and users should carry out their own testing/quality checks when incorporating it in their own system developments. The software when made available as a download, is done so on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied.

As the code evolves each new version has its release date (year, month, day) in its file name as *YYMMDD* with the version, at the time of writing this document, being *240403*.

Each of the functions is described below in some further detail and the aim is, in due course, to publish and maintain subsequent versions of the code at ***this GitHub location***.

In the order in which they appear in the file (which has no significance!):

long int findSize(const char file_name)*

Simple function to get the size of a file where:

- the passed parameter ***file_name*** is the full path/name of the file.

The function returns the size as a number (long int).

---

char copyString(char s[])*

Simple function to copy a string where:

- the passed parameter ***s*** is the string to be copied.

The function returns a copy of the string (char*).

---

void removeString (char text[], int index, int rm_length)

A function to remove characters from a string at a designated position – based upon discussion at:
https://codereview.stackexchange.com/questions/116004/remove-specified-number-of-characters-from-a-string

.. and where:

- the passed parameter ***text*** is the string that is worked on;
- ***index*** is the pointer in the string from which the removal starts;
- ***rm_length*** is the number of characters that are removed.

There is not a returned value, but instead the passed text string is directly modified.

void connect_iot()

This simple 'connection' function, included only for test purposes, just shows in the stdout display the following message:

" *Hello - you are now connected to the 240403 YYMMDD version of the control IoT C functions...*"

- where *240403* is updated to whatever is the current *YYMMDD* version of the code

---

char webpage_download(int debug, const char* domain, const char* page, char* access_token)*

This function downloads a web page contents and some wiki parameters, with the cURL code based upon:

https://curl.se/libcurl/c/getinmemory.html and where:

- the passed parameter ***debug***, if set to 1 produces (lots!!) of additional output;
- ***domain*** is a text string that designates the main URL that must include https:// but no trailing /;
- ***page*** is a text string for the specific web page part of the URL that must include the leading / and spaces 'filled' with %20 NOT + or -
- ***access_token*** is a text string for the Tiki API access token that enables specific permissions for the API usage.

The function returns the web page content and the wiki parameters from the cURL response using a ‘returnstr’ variable.

_Bool webpage_check(int debug, const char domain, const char* page, char* access_token, const char* check_text)*

A function to check web page content looking for specific content on a page and returns 0 if false or 1 if true, with the cURL code based upon:

https://curl.se/libcurl/c/getinmemory.html and where:

- the passed parameter ***debug*** if set to 1 produces (lots!!) of additional output;
- ***domain*** is a text string that designates the main URL that must include https:// but no trailing /;
- ***page*** is a text string for the specific web page part of the URL that must include the leading / and spaces 'filled' with %20 NOT + or -
- ***access_token*** is a text string for the Tiki API access token that enables specific permissions for the API usage;
- ***check_text*** is a text string that is 'looked for' on the web page content.

The function returns either TRUE or FALSE using the variable ‘check_result’.

---

char webpage_datetimecheck(int debug, const char* domain, const char* page, char* access_token, const char* infront_text, int datelen, const char* ref_datetime, const char* datetime_fmt)*

Function to check a web page content looking for a specific date in the content of the page and returns various status text e.g., 'true' if the found date is more recent than the passed check date, or 'false' if not, or various other error status text results.

- the passed parameter ***debug*** if set to 1 produces (lots!!) of additional output;
- ***domain*** is a text string that designates the main URL that must include https:// but no trailing /;
- ***page*** is a text string for the specific web page part of the URL that must include the leading / and spaces 'filled' with %20 NOT + or - ;
- ***access_token*** is a text string for the Tiki API access token that enables specific permissions for the API usage;
- ***infront_text*** is a text string that is used as a marker on the web page that proceeds the date and can be 'looked for' in the web page content;
- ***datelen*** is an integer that is the character length of the date text including any spaces between the date and the infront_text;
- ***ref_datetime***: is a string of the integer Linux epoch time that is being checked against;
- ***datetime_fmt*** is a text string of the expected format of the date text in the web page content e.g. "%a %b %d, %Y %H:%M:%S %Z".

The variable ‘check_result_text’ is used by the function to return the result.

---

*static size_t WriteMemoryCallback(void *contents, size_t size, size_t nmemb, void *userp)*

This function was taken 'as is' from ***https://curl.se/libcurl/c/getinmemory.html*** for use in various other functions where memory is used with call back.

---

*static size_t write_data(void *ptr, size_t size, size_t nmemb, void *stream)*

This function was taken 'as is' from ***https://curl.se/libcurl/c/sepheaders.html*** for use in various File gallery functions.

---

char tracker_item_post(int debug, const char* domain, const char* page, const char* access_token, const char* post_data)*

This function supports the creation of a new Tiki tracker item and uses some code taken from **https://curl.se/libcurl/c/simplepost.html** and where:

- the passed parameter ***debug*** if set to 1 produces (lots!!) of additional output;

- ***domain*** is a text string that designates the main URL that must include https:// but no trailing /;
- ***access_token*** is a text string for the Tiki API access token that enables specific permissions for the API usage;
- ***trackerId*** is a text string of the integer Id of the Tiki tracker that is being 'posted' to;
- ***post_data*** is a text string containing the details of all the fields for the new tracker item e.g. in the format -
  "fields[field01name]=field 01 content&fields[field02name]=field 02 content&etc,etc".
  - where field01name, etc., are the Tiki tracker permanent field names.

The function uses the variable 'returnstr' to return, as a string, the result either as the itemId# of the new tracker item or as a text error message.

---

char tracker_itemupdate(int debug, const char* domain, char* access_token, const char* trackerId, const char* itemId, const char* post_data)*

This function supports the updating of an existing Tiki tracker item and uses some code taken from **https://curl.se/libcurl/c/simplepost.html** and where:

- the passed parameter ***debug*** if set to 1 produces (lots!!) of additional output;
- ***domain*** is a text string that designates the main URL that must include https:// but no trailing /;
- ***access_token*** is a text string for the Tiki API access token that enables specific permissions for the API usage;
- ***trackerId*** is a text string of the integer Id of the Tiki tracker that is being updated;
- ***itemId*** is a text string of the integer Id of the tracker item that is being updated;
- ***post_data*** is a text string containing the details of just the updated tracker item fields e.g. in the format -
  "fields[field01name]=field 01 content&fields[field02name]=field 02 content&etc,etc".
  - where field01name, etc., are the Tiki tracker permanent field names.

The function uses the variable 'returnstr' to return, as a string in a dictionary-like format all the field data for the updated item, or as a text error message.

---

char tracker_itemget(int debug, const char* domain, char* access_token, const char* trackerId, const char* itemId)*

This function supports the downloading of an existing Tiki tracker item and uses some code taken from **https://curl.se/libcurl/c/simplepost.html**.

It should also be noted that a GET mode is not used as this returns too much information, so instead a POST is used with an empty body so that it 'looks like' an 'update' but with nothing updated!

- the passed parameter ***debug*** if set to 1 produces (lots!!) of additional output;
- ***domain*** is a text string that designates the main URL that must include https:// but no trailing /;

- ***access_token*** is a text string for the Tiki API access token that enables specific permissions for the API usage;
- ***trackerId*** is a text string of the integer Id of the Tiki tracker from which the item is being retrieved;
- ***itemId*** is a text string of the integer Id of the tracker item that is being downloaded;

The function uses the variable 'returnstr' to return a string, in a dictionary-like format, containing all the field data for the downloaded item, or as a text error message.

---

char gallery_filedownload(int debug, const char* domain, char* access_token, const char* fileId, const char* filespath, const char* bodyfilename, const char* headerfilename)*

A function to download a file from a Tiki File gallery where:

- the passed parameter ***debug*** if set to 1 produces (lots!!) of additional output;
- ***domain*** is a text string that designates the main URL that must include https:// but no trailing /;
- ***access_token*** is a text string for the Tik1 API access token that enables specific permissions for the API usage;
- fileId is a text string for the integer Id of the file to be downloaded;
- filespath is a text string for the folder path on the calling device (Linux OS assumed) where the downloaded file and the response header file are to be stored, and should include both the first and last / character; and
- both bodyfilename and headerfilename are text string passed parameters for file names in the path defined by filespath and where headerfilename is typically a temporary file on the calling device used for the response headers that would be overwritten with each use, and where bodyfilename can either be 'dictated' by the calling program to be a specific name or if left blank the 'body' i.e. the downloaded file will initially be stored under a temporary name then the actual file name is extracted from the response header file and the downloaded file is renamed.

The function uses the variable 'returnstr' to return, as a text string, the response which is either a 'success' message with details about any local file naming/renaming carried out, or it could be an error message.

---

char gallery_fileupload(int debug, const char* domain, char* access_token, const char* filepath, const char* galId, const char* filename, const char* filetitle, const char* filedesc)*

A function to upload a file to a Tiki File gallery where:

- the passed parameter ***debug*** if set to 1 produces (lots!!) of additional output;
- ***domain*** is a text string that designates the main URL that must include https:// but no trailing /;
- ***access_token*** is a text string for the Tiki API access token that enables specific permissions for the API usage;

- ***filepath*** is a text string for the path/name of the file on the hub device (Linux OS assumed) that is being uploaded;
- ***galId*** is a text string for the integer Id of the Tiki File gallery where the file is to be stored;
- ***filename*** is a text string of just the name of the file without its 'path' details;
- ***filetitle*** is a text string of the short text File gallery title to be assigned to the file;
- ***filedesc*** is a text string of the longer text File gallery description to be assigned to the file;

The function uses the variable 'returnstr' to return, as a text string, the response which is the new Tiki fileId# or it could be an error message.

---

char gallery_fileupdate(int debug, const char* domain, char* access_token, const char* fileId, const char* filepath, const char* filename, const char* filetitle, const char* filedesc)*

A function to update the details of an existing Tiki File gallery file, where:

- the passed parameter ***debug*** if set to 1 produces (lots!!) of additional output;
- ***domain*** is a text string that designates the main URL that must include https:// but no trailing /;
- ***access_token*** is a text string for the Tiki API access token that enables specific permissions for the API usage;
- ***fileId*** is a text string for the fileId# integer for the file that is being updated;
- ***filepath*** is a text string for the full path-name of a file (Linux OS assumed) to replace the existing file - if left blank no change is made;
- ***filename*** is a text string to rename the file that is in Tiki - if left blank no change is made;
- ***filetitle*** is a text string to change the title of the file that is in Tiki - if left blank no change is made;
- ***filedesc*** is a text string to change the description of the file that is in Tiki - if left blank no change is made.

The function uses the variable 'returnstr' to return the response as a text string which, for a successful update, provides all the updated parameters in a dictionary-like format – or it could just be an error message.

Appendix B: *Python test program descriptions*

This set of Python test code should be considered as reasonable working code, although each script contains a lot of (perhaps unnecessary?) debug output and could perhaps be further refined/refactored.

Each item is just a template that needs to be configured for the specific Tiki usage context, i.e. providing individual data for the Tiki web site being accessed, the Tiki API Authorisation code, and other data associated with the test. Each file name includes numerals indicating the version in a *YYMMDD* format, where the current version is *240403*.

The aim is, in due course, to publish and maintain subsequent versions of the code at ***this GitHub location***.

- ***IoT_just_check_web_content_YYMMDD.py*** - checks if some designated text is present in a web page contents, returning a TRUE or FALSE response.
- ***IoT_just_check_web_date_YYMMDD.py*** - looks for a specific date in the content on a web page and returns various status text e.g., 'true' if the found date is more recent than the passed check date, or 'false' if not, or various other error status results.
- ***IoT_just_down_file_YYMMDD.py*** - downloads an existing designated file in a Tiki File gallery, storing it in a designated folder on the hub device, where optionally the stored filename can either be explicitly specified or the original name of the file when it was uploaded to the gallery is extracted and used.
- ***IoT_just_download_web_page_YYMMDD.py*** - downloads and returns to the Python code the full cURL response which normally is the contents of a web page plus various wiki parameters, but could be an error message!
- ***IoT_just_get_tracker_item_YYMMDD.py*** - gets the details of an existing Tiki tracker item.
- ***IoT_just_up_file_YYMMDD.py*** - uploads a file to a designated Tiki File gallery.
- ***IoT_just_update_file_YYMMDD.py*** - updates the various elements of an existing file in a Tiki File gallery.
- ***IoT_just_update_T_opcode_YYMMDD.py*** - uses the **tracker_itemget** and the ***tracker_itemupdate*** functions to demonstrate how the "Remote execution (4)" usage scenario (see **here**) might be implemented.
- ***IoT_just_update_tracker_item_YYMMDD.py*** - updates the details of an existing Tiki tracker item and returns the 'mes' text from the cURL response which would normally be something like "*Tracker item NN has been updated*" where NN is the tracker itemId# that is being updated. Alternatively error texts such as "*Success text not found*" or "*mes text not found*" might be returned.
- ***IoT_just_upload_new_tracker_item_YYMMDD.py*** - uploads a new tracker item to an existing Tiki tracker and either returns the new tracker itemId# or if there is an error it returns "*itemId text not found*"

Appendix C: IoT_Ctest_template_240403.c

This example code provides a demonstration of what is possible when compiling a 100% 'C' code routine that uses the Tiki API access functions in *control_iot_YYMMDD.c*

The aim is, in due course, to publish and maintain subsequent versions of the code at ***this GitHub location***.

It is provided as a documented template that needs to be configured for the specific Tiki site and functions it is to be used for.

Its primary purpose is to simply demonstrate how the various C functions described in Appendix A can be 'called from a main program written in C.

As the code evolves each new version has its release date (year, month, day) in its file name as YYMMDD

END OF DOCUMENT