Middleware for UECS Nodes

UARDECS_ESP32C3 Ver. 1.0.x Essential Guide

Author: Takehiko Hoshi

October 14, 2026 Edition

Contents

1. Introduction 2
2. Supported Boards and Precautions for Using the ESP32C3 3
  2.1 Safe Mode Pin and Wi-Fi Connection Precautions 3
  2.2 Precautions for Using the CPU Board GPIO Pins 4
  2.3 Precautions When Using the ESP32C3 CPU 5
  2.4 Precautions Specific to the ESP32C3 Version 5
3. Configuring UARDECS_ESP32C3 in the Arduino IDE 6
4. Safe Mode and Normal Mode on the ESP32C3 Version 7
5. Learning to Use the Library Through the Sample Programs 7
6. UARDECS_ESP32C3 Variables and Functions (Reference) 8
  6.1 Required Global Constants and Variables 8
  6.2 Required Sketch Functions 10
  6.3 Optional Sketch Functions 12
7. Conclusion 12

1. Introduction

UARDECS (pronounced "you-are-decks") for ESP32C3 is middleware developed to create nodes for the Ubiquitous Environment Control System (UECS, https://uecs.jp) using boards equipped with the small, low-cost ESP32C3 CPU and the freely available Arduino IDE development environment. In UECS, nodes share information and coordinate with one another over an open communication platform called UECS-CCM (Common Communication Message). This middleware implements communication compliant with UECS-CCM and simplifies the development of UECS-compliant nodes. The UECS-CCM communication protocol document can be downloaded from the UECS Society at https://www.uecs.jp/outline/kiyaku.

Although the ESP32C3 has relatively few I/O pins, it includes Wi-Fi and Bluetooth and enables IoT functionality on an ultra-compact board only a few centimeters square. Boards cost roughly several hundred to one thousand yen, making it possible to embed one in each sensor or actuator and achieve the original concept of true autonomous distributed control. This middleware makes it easy to port software assets for UECS nodes developed with the Arduino IDE on Arduino and Raspberry Pi Pico. The implemented UECS-CCM (Common Communication Message) protocol follows the same specifications as the UARDECS_MEGA version.

This middleware is released as open source on GitHub at https://github.com/uecs-star/UARDECS_ESP32C3. It can also be installed from the Arduino IDE Library Manager by searching for "UARDECS_ESP32C3." This library is provided under the following license.

『

The Arduino IDE library software "UARDECS_ESP32C3" is
distributed under the 2-Clause BSD License.

Copyright 2026 Takehiko Hoshi, Hideto Kurosaki and Ken-ichiro Yasuba,
Wakayama, Japan

Redistribution and use in source and binary forms,
with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions,
and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions,
and the following disclaimer in the documentation and/or other materials provided
with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The authors developed this middleware with assistance from GPT Codex 6.1 Sol, using the UARDECS_PICO version as a template. The middleware originated in "Development of a Program Library for Building Low-Cost Environmental Control Systems" (Ken-ichiro Yasuba et al., 2016), Agricultural Information Research, Vol. 25, No. 1, pp. 12-18 (https://doi.org/10.3173/air.25.12).

2. Supported Boards and Precautions for Using the ESP32C3

This middleware has been verified with Arduino IDE 2.3.10. Operation has been confirmed on the ESP32C3-OLED board fitted with a 0.42-inch display and on the XIAO ESP32C3. Compatible products and other ESP32C3-based boards should also work when the appropriate board is selected in the Arduino IDE. However, operation is not guaranteed; use them at your own risk. Precautions for the two verified boards are given below.

2.1 Safe Mode Pin and Wi-Fi Connection Precautions

(a) Safe Mode (the mode used for the initial DHCP connection) is selected by the connection state of GPIO 3. On the XIAO ESP32C3, D1 on the board is GPIO 3 (note that D3 on the board is GPIO 5). Connecting D1 to GND (0 V) enables Safe Mode. Leaving it open enables Normal Mode, which connects using the IP address configured on the web page.

(b) The ESP32C3 uses its built-in Wi-Fi interface for network communication. In setup(), enter the SSID and password of an available Wi-Fi network in setWiFiCredentials("SSID", "Password"); In Safe Mode, these credentials are used to connect by DHCP (automatic address assignment).

(c) In Normal Mode, the connection uses the SSID, IP address, and other information entered under Network Config on the web page. Configure this information correctly in Safe Mode, and then leave the GPIO 3 pin (D1 on the XIAO) open.

(d) The built-in Wi-Fi chip already has a MAC (physical) address, so the setting required in earlier versions of UARDECS is no longer needed. To obtain the MAC address, call WiFi.macAddress().

2.2 Precautions for Using the CPU Board GPIO Pins

***ESP32C3+0.42inchOLED

· GPIO2: One of the strapping pins used to select the boot mode. If a circuit that changes the pin voltage at power-up (such as a strong pull-down) is connected, the board may fail to boot normally. After startup, it can be used as a normal GPIO pin.

· GPIO5: Connected to the OLED SDA (data line).

· GPIO6: Connected to the OLED SCL (clock line).

· GPIO8: Connected to the on-board blue LED.

· GPIO9: Connected to the BOOT button. Because this is an important pin for selecting the boot mode, wiring an external circuit that holds it at GND can prevent program upload or normal startup.

· GPIO12 to GPIO17: Reserved for the built-in SPI flash memory. Because these pins continuously perform high-speed communication to read the program, they cannot be used as general-purpose GPIO pins.

· GPIO18 and GPIO19: Used by the USB-JTAG / USB serial functions for program upload through the USB Type-C port and for serial monitor communication. They can be used as GPIO pins if the function is disabled in software, but doing so risks losing communication with the PC and is best avoided.

***XIAO ESP32C3

· D2 (GPIO2): This is also a strapping pin associated with the boot mode. Because there is no internal pull-up or pull-down, the connected device may cause startup to fail.

· D3 (GPIO5 / A3, etc.): In some environments, such as older Arduino IDE board definitions, the pin may not be recognized or defined correctly as analog input A3, and Wi-Fi/BLE communication may affect analog readings. It is also shared with a JTAG signal line such as MTDI.

· D4 and D5 (GPIO6 and GPIO7) are used as the I2C (Wire) SDA and SCL lines, so it is safest not to use them for other purposes.

· D6 is configured as a UART output at startup. If it is used as an input, an unexpectedly high current may flow. Use it as an output.
· D8 (GPIO8): One of the strapping pins that select the boot mode. If it is LOW at startup while GPIO9 is also LOW, the board may behave unexpectedly or enter a special mode (UART_BOOT). When connecting external circuitry, keep the startup voltage at the appropriate level, such as HIGH.
· D9 (GPIO9): Connected to the on-board BOOT button and a pull-up resistor. It is well suited for use as a switch input, but a circuit that pulls it to GND at startup or presents a heavy external load can prevent normal startup and entry into download mode.

The operating voltage is 3.3 V. According to the official data sheet, when drive strength is set to its maximum (PAD_DRIVER = 3), each GPIO pin can source up to 40 mA and sink up to 28 mA.

2.3 Precautions When Using the ESP32C3 CPU

(1) GPIO 3 is used as the signal pin for selecting Safe Mode. On the XIAO ESP32C3 board, this is pin D1. Connecting it to GND (0 V) enables Safe Mode. UARDECS checks the pin state only once during setup; changing it afterward has no effect.
(2) On the ESP32, data declared with const is automatically allocated to flash memory, so PROGMEM= is no longer required.
(3) The ESP32C3 is assigned a MAC address in advance, so it does not need to be defined. Because the connection uses Wi-Fi, the SSID and password for Safe Mode must be configured. To conserve memory, each string is limited to 25 characters; values longer than 25 characters cannot be set. In Safe Mode, the board connects using DHCP and receives an address automatically.

2.4 Precautions Specific to the ESP32C3 Version

(1) The connection has changed from Ethernet to Wi-Fi, with corresponding specification changes. It has been reported that Wi-Fi-specific security settings, such as client isolation, can prevent broadcast CCMs from reaching other nodes. Review the security settings of the Wi-Fi bridge or router and verify packet delivery with a network analyzer or similar tool. If issues persist after multiple attempts, restarting devices such as a Wi-Fi bridge may be effective.
(2) The provider-response behavior for the TYPE attribute of provider-search CCMs sent over UDP port 16521 has changed. Previously, only providers whose TYPE attribute exactly matched the search key responded. In the ESP32C3 version, providers with a partial match also respond. This makes it easy to find providers with the same item name or node type name under the CCM identifier naming

convention. Applications that use this protocol will receive response CCMs from more providers, so take care to avoid buffer overflow.

(3) As a substitute for the EEPROM used on Arduino boards, this library uses 4,096 bytes of unused ESP32C3 program flash as nonvolatile storage. It also uses 4,096 bytes of SRAM as a cache allocated after startup, reducing the memory available to the application by that amount.

The nominal rewrite endurance is the same as Arduino EEPROM (100,000 write cycles), but flash can be rewritten only in 4,096-byte units. Therefore, even a small change consumes endurance across the entire 4,096-byte area. This may not seriously affect infrequently updated settings such as IP addresses, but it is highly disadvantageous for applications that write at regular intervals, such as data logging. The limit will probably be reached far sooner than with Arduino EEPROM, which can be rewritten byte by byte. If the program rewrites EEPROM frequently, use a separate device capable of storing nonvolatile data.

3. Configuring UARDECS_ESP32C3 in the Arduino IDE

(a) Install the ESP32 board package

Open [File] > [Preferences]. Enter the following URL in [Additional Boards Manager URLs], and then click [OK]: https://githubusercontent.com

(b) Install the ESP32 board

Open [Tools] > [Board] > [Boards Manager]. Enter esp32 in the search box, and install "esp32" by Espressif Systems.

(c) Connect the computer to the ESP32-C3 board with a USB cable.

Under [Tools] > [Board] > [ESP32 Arduino], select the board in use (for example, ESP32C3 Dev Module or XIAO_ESP32C3). Then select the connected [Serial Port].

(d) Set USB CDC On Boot to Enabled

Boards whose USB connector is wired directly to ESP32-C3 GPIO18/19, including the XIAO and many 0.42-inch OLED boards, cannot display Serial.print() output in the console after compilation and upload unless USB CDC On Boot is set to Enabled. Select [Tools] > [USB CDC On Boot: Enabled]. For boards fitted with a USB-to-UART converter such as a CP210x or CH340, select Disabled.

The environment is now ready for compilation. Repeat steps (c) and (d) whenever you replace the board, even with another board of the same model.

(e) Click the arrow icon at the upper left of the IDE to compile and upload the program.

(f) When the upload is complete, the board resets automatically and runs the program.

4. Safe Mode and Normal Mode on the ESP32C3 Version

UARDECS provides a Safe Mode for configuring the node's IP connection from a web page without rewriting the program. To enable Safe Mode, connect GPIO3, or terminal D1, to GND (0 V) before startup. Because this version communicates over Wi-Fi, its Safe Mode specifications differ from those of earlier UARDECS versions as described below.

In Safe Mode, setup() must call setWiFiCredentials(SSID, PASS); before UECSsetup(); and specify the SSID and PASSWORD strings for the network to be used. Each string in UARDECS is limited to 25 characters, which is shorter than the maximum allowed by the Wi-Fi standard. In the sample programs, "SSID" and "PASSWORD" are defined near the beginning with #define statements. The IP address is obtained automatically by DHCP. It is convenient to use a DHCP-enabled Wi-Fi router on the same network as the development PC. A dedicated Wi-Fi router for development can improve efficiency when developing many nodes.

The connected IP address appears in the serial monitor. Open http://<connected-IP-address>/ in a browser to display the configuration page, where you can set the Wi-Fi network and IP address for actual operation. Then leave GPIO3, or terminal D1, open (pulled up or unconnected) and reset the node to enter Normal Mode.

5. Learning to Use the Library Through the Sample Programs

The UARDECS library folder contains an examples folder with two sample programs. If the UARDECS library has been registered correctly in the Arduino IDE, load and run the two samples from [File] > [Examples] > [UARDECS_ESP32C3]. The comments in these sketches explain the basic usage of UARDECS_ESP32C3.

The sample programs are:

(1) UECS-CCMtransmitter.ino: A CCM transmission example that outputs two types of CCMs, one for sensor measurements (every 10 seconds) and one for remote operation (every second). Because the demonstration runs on a board without sensors or switches, all values can be entered and tested through the web interface (HTTP).

(2) UECS-CCMreceiver.ino: A sample program that receives sensor measurements (every 10 seconds) and remote-control values (every second), and performs ON/OFF control according to a setpoint. Because the demonstration runs on a board without sensors or switches, all values can be entered and tested through the web interface (HTTP).

Upload these programs to separate boards, connect the boards to the same Wi-Fi network segment, and set matching TYPE, ROOM, REGION, and ORDER attributes for their CCMs. You can then learn how to send sensor measurements from UECS-CCMtransmitter, receive them with UECS-CCMreceiver, and switch an output on and off so that the value does not fall below the setpoint. You can also learn how to send manual/automatic remote-control data from UECS-CCMtransmitter and apply the received value to an output on UECS-CCMreceiver.

To inspect the CCMs being exchanged, use the UECS Packet Transmit/Receive Tool for Windows PCs, a CCM packet analyzer. It is available at https://hoshi-lab.info/uecs.org/arduino/uecsrs.html or https://github.com/H-Kurosaki/UecsRS. A warning may appear the first time it starts; allow it to run.

6. UARDECS_ESP32C3 Variables and Functions (Reference)

6.1 Required Global Constants and Variables

const char - U_name[] - Device model name (for example, "Temp controller"). Use no more than 20 single-byte alphanumeric characters. Tags and double quotation marks are not allowed. This becomes the initial title of the web page.

const char - U_vender[] - Device developer (for example, "XXX co."). Use no more than 20 single-byte alphanumeric characters. Tags and double quotation marks are not allowed.

const char - U_uecsid[] - ID number issued by the UECS Society, represented as a hexadecimal string. It must be exactly 12 single-byte alphanumeric characters and is used in responses to NODESCAN. Prototypes may use "000000000000".

const char - U_footnote[] - Text displayed at the bottom of the top page when the node's HTTP server is accessed. There is no length limit, and Japanese text may be used. Tags are displayed as entered, so unnecessary tags can disrupt the page layout; they can also be used intentionally to add links.

char - U_nodename[20] - Node name displayed by the node's HTTP server. Use no more than 19 single-byte alphanumeric characters. Japanese text may be used in UTF-8 up to six characters. The characters "&<>" are not allowed. This string is the initial value in Safe Mode. The user can change it, and the value is stored in nonvolatile memory.

const int - U_HtmlLine - Total number of configuration items, such as choices, displayed on the Node Status page of the node's HTTP server. Specify 0 if no items are used.

struct UECSUserHtml - U_html[U_HtmlLine] - Structure for the variables displayed and configured on the web page. Define each array element as {Name, Form format, Unit, Remark, Select, ItemN, value, min, max, digit}, as shown in the sample below.

```
『   // setting CCM description for web page
     const int U_HtmlLine = 2; //Total number of HTML table rows.
     // No.1 CCM item
     const char NAME1_st[] = "Meas. value";
     const char UNIT1_st[] = "";
     const char NOTE1_st[] = "Sensor value";
     signed long MeasVal;
     // No.2 CCM item
     const char NAME2_st[] = "Remote SW";
     const char UNIT2_st[] = "";
     const char NOTE2_st[] = "Remote control";
     signed long RemoSW;
     const char SW_auto[] = "Auto";
     const char SW_off[] = "OFF";
     const char SW_on[] = "ON";
     const char *Menu_rsw[3] = { SW_auto, SW_off, SW_on } ;
     const char** dummy = NULL;
     // How to show any items in Web page.
     struct UECSUserHtml U_html[U_HtmlLine] = {
          // Name, Form format, Unit, Remark, Select,ItemN, value, min, max, digit
          {NAME1_st, UECSINPUTDATA,    UNIT1_st, NOTE1_st, dummy, 0, &(MeasVal),
          -1000, 1000, 1},
          {NAME2_st, UECSSELECTDATA, UNIT2_st, NOTE2_st, Menu_rsw, 3,
          &(RemoSW), 0, 2, 0},
     };
                                                                          』
```

const int U_MAX_CCM - Specifies the number of CCMs used.

struct UECSCCM - U_ccmList[U_MAX_CCM] - It is convenient to define the IDs of the CCMs to use as an enum, add CCMID_dummy at the end, and then define U_MAX_CCM and the UECSCCM structure as shown below.

『

```
enum {
        CCMID_MeasVal,
       CCMID_RemoteSW,
       CCMID_cnd,
        CCMID_dummy
       };
const int U_MAX_CCM = CCMID_dummy;
UECSCCMU_ccmList[U_MAX_CCM];
```

』

struct UECSOriginalAttribute - U_orgAttribute - Variable that records the communication state of this UECS node. Declare it near the beginning of the sketch as a global variable without changing the variable name, as follows:

```
UECSOriginalAttribute U_orgAttribute;
```

6.2 Required Sketch Functions

void UserInit() - Initializes in the sketch the CCMs declared in struct UECSCCM U_ccmList[U_MAX_CCM]. Define each CCM inside this function by calling the UECSsetCCM() function described next.

void UECSsetCCM() - Defines the attributes of each CCM. Implement it for the number specified by U_MAX_CCM. After describing the values as global constants, call it from UserInit() as shown in the sample below.

『

```
const char ccmName1[] = "Measurement value";
const char ccmType1[] = "MeasVal.mXX";
const char ccmUnit1[] = "";
const char ccmName2[] = "Remote Switch";
const char ccmType2[] = "RemoSW.mXX";
const char ccmUnit2[] = "";
const char ccmNameCnd[] = "NodeCondition";
```

```
const char ccmTypeCnd[] = "cnd.mXX";
const char ccmUnitCnd[] = "";
// Description of CCM communication
void UserInit() { // S/!R is true for a transmitted CCM and false for a received CCM
    // S/!R, Index name, Description, Type name, unit, priority, digit, CCM Lv.
    UECSsetCCM(true, CCMID_MeasVal, ccmName1, ccmType1, ccmUnit1, 29, 1, A_1M_1);
    UECSsetCCM(true, CCMID_RemoteSW, ccmName2, ccmType2, ccmUnit2, 29, 0,
    A_1S_0);
    UECSsetCCM(true, CCMID_cnd, ccmNameCnd, ccmTypeCnd,ccmUnitCnd,29, 0,
    A_1S_0);
}
```

Regarding CCM levels, the shortest processing interval (tick) in this implementation is one second. Therefore, even if a demand such as a value change occurs at the A_10S_1 or A_1M_1 level, the delay may be as long as one second. In addition, A_1S_0 and A_1S_1 are equivalent, and B_0 and B_1 are not supported. For freshness control, the period for which a CCM remains valid after reception stops is set to three times its transmission interval (1 second, 10 seconds, or 1 minute).

void OnWebFormRecieved() - Called when the Send button is pressed on the Node Status page generated by the HTTP server and values such as selected options are submitted. After this function runs, the values are stored in flash memory. Even when no processing is required, this function must be implemented with an empty body.

void UserEverySecond() - Called once per second; place processing that must run at one-second intervals in this function. After it runs, messages that meet their transmission conditions are sent to port 16520. Even when no processing is required, this function must be implemented with an empty body.

void UserEveryMinute() - Called every 60 seconds; place processing that must run at 60-second intervals in this function. Even when no processing is required, this function must be implemented with an empty body.

void UserEveryLoop() - Called after the system timer count, HTTP server processing, and messages on UDP ports 16520, 16529, and 16521 have been checked. Because it is called frequently, do not place time-consuming processing in it. Even when no processing is required, this function must be implemented with an empty body.

void UECSloop() - Must be called from void loop(), which performs normal processing in an Arduino IDE sketch. Frequently repeated node processing can also be placed there, but as with void UserEveryLoop(), time-consuming processing can destabilize the periodic UECS tasks and should not

be placed in void loop(). Whenever possible, put such processing in void UserEverySecond() and limit loop() to lightweight work such as checking key presses.

void setWiFiCredentials("SSID", "PASSWORD") - Credentials for the Safe Mode Wi-Fi connection. This call must appear in void setup(), where the Arduino IDE sketch performs initialization, and before void UECSsetup() is executed. Pass each argument as a char* string pointer. Each string can contain up to 25 characters.

void UECSsetup() - UARDECS initialization function. It must be called from void setup() after the sketch performs any required initialization.

6.3 Optional Sketch Functions

char* getUARDECSfor() - Returns a string identifying the CPU targeted by this UARDECS version. This version returns "UARDECS_ESP32C3".

char* getUARDECSon() - Returns the version string for this UARDECS library.

char* getUECSWiFiSSID() - Returns the SSID string of the connected Wi-Fi network.

char* getUECSWiFiPASSWORD() - Returns the password (passphrase) of the connected Wi-Fi network. To prevent information leakage in a production product, use the returned string only for internal processing and do not expose it in readable form.

bool UECSWiFiConnected() - Returns the Wi-Fi connection state: true when connected and false when disconnected.

bool UECSSafeMode() - Returns true in Safe Mode and false in Normal Mode.

void WiFi.setSleep(bool) - The ESP32C3 can operate Wi-Fi intermittently to save power. Because UECS must communicate over Wi-Fi at intervals as short as one second, calling WiFi.setSleep(false); before Wi-Fi communication begins, and before UECSsetup(); in UARDECS, stabilizes communication. To use UECS with intermittent communication, such as in battery-powered operation, call this function with true and implement the sketch accordingly.

7. Conclusion

This guide provides the minimum information required to use UARDECS_ESP32C3. For details not covered here, refer to the UARDECS_MEGA and UARDECS_PICO manuals at https://github.com/H-Kurosaki/UARDECS_Documents and to the sample programs included with this middleware.