

Module USB v1.2 Arduino Guide

1. Preparation

- 1. Environment setup: Follow the [Arduino IDE guide](#) to install the IDE, then install the board package for the board you are using.
- 2. Required libraries:
 - [M5Unified](#)
 - [M5GFX](#)
 - [M5-Max3421E-USBSHield](#)
- 3. Required hardware:
 - [CoreS3](#)
 - [Module USB v1.2](#)
 - USB mouse
 - USB keyboard



2. Notes

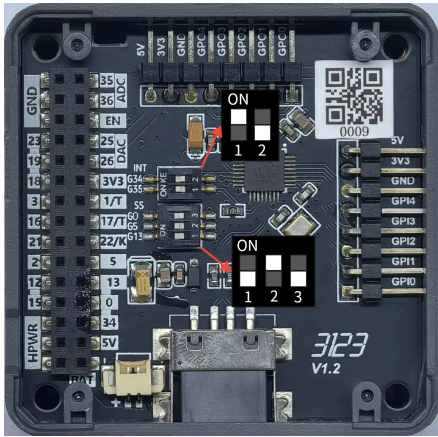
Pin Compatibility

Each host has a different pin configuration. Before use, check the [pin compatibility table](#) in the product documentation and modify the example according to the actual pin connections.

3. Example Programs

- This tutorial uses CoreS3 as the host device with Module USB v1.2 to provide USB host functionality. Module USB v1.2 communicates with the host over SPI. Adjust the pin definitions in the example to match the actual circuit connections. The corresponding SPI I/O after connection is [G36 \(SCK\)](#) , [G37 \(MOSI\)](#) , [G35 \(MISO\)](#) , [G1 \(CS\)](#) , and [G10 \(INT\)](#) .

Set the DIP switches as shown below:



3.1 USB Mouse

This example uses the MAX3421E controller on Module USB v1.2 in USB host mode. Connect a USB mouse to the module's USB port. The top of the CoreS3 screen shows the connection status, the center shows the mouse activity area and pointer, and the bottom shows button, movement, coordinate, and latest-status information. The program updates the interface and outputs HID reports over the serial port as you move or click the mouse.

cpp

```

1  #include <M5Unified.h>
2  #include <M5_Max3421E_Usb.h>
3
4  // CoreS3: SCK=36, MOSI=37, MISO=35, CS=1, INT=10.
5  M5_USBH_Host USBHost(&SPI, 36, 37, 35, 1, 10);
6  M5Canvas mouse_canvas(&M5.Display);
7
8  // Track the pointer, button state, last movement, and connection status.
9  static int cursor_x = 160;
10 static int cursor_y = 120;
11 static uint8_t mouse_buttons = 0;
12 static int8_t mouse_dx = 0;
13 static int8_t mouse_dy = 0;
14 static bool mouse_connected = false;
15 static const char* mouse_status = "WAITING FOR MOUSE";
16
17 void drawPointer()
18 {
19     // Draw a small pointer marker inside the activity area.
20     mouse_canvas.drawLine(cursor_x, cursor_y, cursor_x, cursor_y + 10, WHITE);
21     mouse_canvas.drawLine(cursor_x, cursor_y, cursor_x + 7,
22                             cursor_y + 7, WHITE);
23     mouse_canvas.drawLine(cursor_x + 4, cursor_y + 7, cursor_x + 7,
24                             cursor_y + 7, WHITE);
25     mouse_canvas.drawLine(cursor_x + 4, cursor_y + 7, cursor_x,
26                             cursor_y + 10, WHITE);
27     mouse_canvas.drawLine(cursor_x + 3, cursor_y + 7, cursor_x + 6,
28                             cursor_y + 12, WHITE);
29 }
30
31 void renderMouseUI()
32 {
33     // Redraw the complete screen from the current mouse state.
34     mouse_canvas.fillSprite(BLACK);
35     mouse_canvas.setTextColor(WHITE, BLACK);
36     mouse_canvas.setTextSize(2);
37     mouse_canvas.setTextWrap(false, false);
38
39     mouse_canvas.setCursor(8, 5);
40     mouse_canvas.print("USB MOUSE");
41     mouse_canvas.setCursor(248, 5);
42     mouse_canvas.print(mouse_connected ? "READY" : "WAIT");
43     mouse_canvas.drawLine(0, 28, 319, 28, WHITE);
44
45     // Keep the pointer inside the bordered activity area.
46     mouse_canvas.drawRect(0, 34, 319, 204, WHITE);
47     drawPointer();
48     if (mouse_buttons & 0x01) {
49         mouse_canvas.fillCircle(cursor_x, cursor_y, 3, WHITE);
50     }
51     if (mouse_buttons & 0x02) {
52         mouse_canvas.fillCircle(cursor_x, cursor_y, 3, GREEN);
53     }
54
55     mouse_canvas.drawLine(0, 209, 319, 209, WHITE);

```

```

56     mouse_canvas.setTextSize(1.25f);
57     mouse_canvas.setCursor(8, 212);
58     mouse_canvas.printf("BTN 0x%02X", mouse_buttons);
59     mouse_canvas.setCursor(132, 212);
60     mouse_canvas.printf("DX %d DY %d", mouse_dx, mouse_dy);
61     mouse_canvas.setCursor(8, 226);
62     mouse_canvas.printf("%s", mouse_status);
63     mouse_canvas.setCursor(180, 226);
64     mouse_canvas.printf("X%03d Y%03d", cursor_x, cursor_y);
65     mouse_canvas.pushSprite(0, 0);
66 }
67
68 void showMouseStatus(const char* status, uint8_t buttons = 0,
69                     int16_t move_x = 0, int16_t move_y = 0)
70 {
71     // Update the values shown in the footer before rendering.
72     mouse_status = status;
73     mouse_buttons = buttons;
74     mouse_dx = move_x;
75     mouse_dy = move_y;
76     renderMouseUI();
77 }
78
79 void movePointer(int16_t move_x, int16_t move_y)
80 {
81     // Clamp movement so the pointer never overwrites the border.
82     cursor_x = constrain(cursor_x + move_x, 8, 308);
83     cursor_y = constrain(cursor_y + move_y, 42, 190);
84 }
85
86 void setup()
87 {
88     // Initialize the display, serial port, sprite, and USB host.
89     M5.begin();
90     Serial.begin(115200);
91     mouse_canvas.createSprite(320, 240);
92     showMouseStatus("Waiting for mouse...");
93
94     Serial.println("M5Module USB HOST HID");
95     if (USBHost.begin(0)) {
96         Serial.println("MODULE INIT OK");
97         showMouseStatus("USB host ready");
98     } else {
99         Serial.println("MODULE INIT ERROR");
100        showMouseStatus("USB host error");
101    }
102 }
103
104 void loop()
105 {
106     // Poll the USB host so HID callbacks can receive reports.
107     USBHost.task();
108     delay(1);
109 }
110

```

```

111 void tuh_hid_report_sent_cb(uint8_t dev_addr, uint8_t instance,
112                             uint8_t const* report, uint16_t len)
113 {
114     // This host example does not send HID reports.
115     (void)dev_addr;
116     (void)instance;
117     (void)report;
118     (void)len;
119 }
120
121 // Invoked when a USB HID device is connected.
122 void tuh_hid_mount_cb(uint8_t dev_addr, uint8_t instance,
123                       uint8_t const* desc_report, uint16_t desc_len)
124 {
125     (void)desc_report;
126     (void)desc_len;
127
128     uint16_t vid;
129     uint16_t pid;
130     tuh_vid_pid_get(dev_addr, &vid, &pid);
131     Serial.printf("HID mounted: address=%d, instance=%d, VID=%04X, PID=%04X\r\n",
132                  dev_addr, instance, vid, pid);
133     mouse_connected = true;
134     showMouseStatus("Mouse connected");
135
136     if (!tuh_hid_receive_report(dev_addr, instance)) {
137         Serial.println("Cannot request HID report");
138     }
139 }
140
141 // Invoked when a USB HID device is disconnected.
142 void tuh_hid_umount_cb(uint8_t dev_addr, uint8_t instance)
143 {
144     Serial.printf("HID unmounted: address=%d, instance=%d\r\n",
145                  dev_addr, instance);
146     mouse_connected = false;
147     showMouseStatus("Mouse disconnected");
148 }
149
150 // Invoked when a HID report is received from the interrupt endpoint.
151 void tuh_hid_report_received_cb(uint8_t dev_addr, uint8_t instance,
152                                 uint8_t const* report, uint16_t len)
153 {
154     if (len >= 3) {
155         // A boot mouse report starts with buttons, X movement, and Y movement.
156         // Button bits are left=0x01, right=0x02, and middle=0x04.
157         const uint8_t buttons = report[0];
158         const int8_t move_x = static_cast<int8_t>(report[1]);
159         const int8_t move_y = static_cast<int8_t>(report[2]);
160
161         movePointer(move_x, move_y);
162         showMouseStatus("Mouse report", buttons, move_x, move_y);
163         Serial.printf("HID report: buttons=0x%02X, dx=%d, dy=%d\r\n",
164                      buttons, move_x, move_y);
165     }

```

```

166         // Use the middle button to clear the canvas.
167         if (buttons & 0x04) {
168             mouse_canvas.fillSprite(BLACK);
169             mouse_canvas.pushSprite(0, 0);
170         }
171     }
172
173     if (!tuh_hid_receive_report(dev_addr, instance)) {
174         Serial.println("Cannot request next HID report");
175     }
176 }

```

After the program starts and the module initializes successfully, the CoreS3 screen shows the USB host and device status, and the serial monitor outputs information similar to the following:

```

M5Module USB HOST HID
MODULE INIT OK
HID mounted: address=1, instance=0, VID=XXXX, PID=XXXX
HID report: buttons=0x00, dx=1, dy=0

```

The pointer moves on the CoreS3 screen as you move the mouse. Pressing the left or right mouse button draws a white or green marker at the pointer position, respectively. Pressing the middle button clears the screen.



3.2 USB Keyboard

This example uses the MAX3421E controller on Module USB v1.2 in USB host mode. After connecting a USB keyboard, the CoreS3 screen displays white input text on a black background, the cursor, and keyboard status. The serial port outputs HID reports and the decoded characters.

cpp

```

1  #include <M5Unified.h>
2  #include <M5_Max3421E_Usb.h>
3
4  // CoreS3: SCK=36, MOSI=37, MISO=35, CS=1, INT=10.
5  M5_USBH_Host USBHost(&SPI, 36, 37, 35, 1, 10);
6  M5Canvas canvas(&M5.Display);
7
8  // Store text editing state and the current keyboard status.
9  String typed_text;
10 String last_key = "--";
11 uint8_t previous_keys[6] = {0};
12 bool caps_lock = false;
13 bool num_lock = false;
14 bool keyboard_connected = false;
15 uint16_t cursor_position = 0;
16
17 static constexpr uint16_t MAX_TEXT_LENGTH = 240;
18 static constexpr uint16_t MAX_LAYOUT_LINES = MAX_TEXT_LENGTH + 1;
19 static constexpr int16_t INPUT_LEFT = 8;
20 static constexpr int16_t INPUT_TOP = 42;
21 static constexpr int16_t INPUT_RIGHT = 312;
22 static constexpr int16_t INPUT_LINE_HEIGHT = 16;
23 static constexpr uint8_t INPUT_MAX_LINES = 10;
24
25 // Each entry stores the character range displayed on one visual line.
26 struct InputLine {
27     uint16_t start;
28     uint16_t end;
29 };
30
31 InputLine input_lines[MAX_LAYOUT_LINES];
32 uint16_t input_line_count = 1;
33
34 bool isNewKey(uint8_t key)
35 {
36     // Process a key only once while it remains pressed.
37     for (uint8_t previous : previous_keys)
38         if (previous == key) return false;
39     return true;
40 }
41
42 char keyToAscii(uint8_t key, bool shifted)
43 {
44     // Convert standard HID keyboard usages to printable characters.
45     if (key >= 0x04 && key <= 0x1D) {
46         char value = 'a' + key - 0x04;
47         if (shifted != caps_lock) {
48             value = value - 'a' + 'A';
49         }
50         return value;
51     }
52
53     if (key >= 0x1E && key <= 0x27) {
54         static const char normal[] = "1234567890";
55         static const char shifted_values[] = "!@#$%^&*()";

```

```

56         return shifted ? shifted_values[key - 0x1E] : normal[key - 0x1E];
57     }
58
59     if (key == 0x2C) return ' ';
60     if (key >= 0x2D && key <= 0x38) {
61         static const char normal[] = "-=[]\\0;\x60,./";
62         static const char shifted_values[] = "_+{}|\0:\\"~<>?";
63         return (shifted ? shifted_values : normal)[key - 0x2D];
64     }
65     return key >= 0x54 && key <= 0x57 ? "/*-+"[key - 0x54] : 0;
66 }
67
68 char keypadToAscii(uint8_t key)
69 {
70     // Convert keypad number keys only when NumLock is active.
71     if (!num_lock) return 0;
72     if (key >= 0x59 && key <= 0x61) return '1' + key - 0x59;
73     return key == 0x62 ? '0' : key == 0x63 ? '.' : 0;
74 }
75
76 void limitTextLength()
77 {
78     // Keep the buffer bounded and keep the cursor within the buffer.
79     if (typed_text.length() > MAX_TEXT_LENGTH) {
80         const uint16_t removed =
81             typed_text.length() - MAX_TEXT_LENGTH;
82         typed_text.remove(0, removed);
83         if (cursor_position >= removed) {
84             cursor_position -= removed;
85         } else {
86             cursor_position = 0;
87         }
88     }
89     if (cursor_position > typed_text.length()) {
90         cursor_position = typed_text.length();
91     }
92 }
93
94 void insertText(const String& value)
95 {
96     // Insert text at the cursor instead of replacing the whole buffer.
97     String updated = typed_text.substring(0, cursor_position);
98     updated += value;
99     updated += typed_text.substring(cursor_position);
100     typed_text = updated;
101     cursor_position += value.length();
102     limitTextLength();
103 }
104
105 void buildInputLayout()
106 {
107     // Rebuild visual lines for explicit newlines and automatic wrapping.
108     canvas.setTextSize(2);
109     const int32_t text_width = INPUT_RIGHT - INPUT_LEFT - canvas.textWidth("_");
110     const uint16_t length = typed_text.length();

```

```

111     String current_line;
112
113     input_line_count = 1;
114     input_lines[0] = {0, 0};
115
116     for (uint16_t i = 0; i < length; ++i) {
117         const char value = typed_text[i];
118         if (value == '\n') {
119             input_lines[input_line_count - 1].end = i;
120             if (input_line_count < MAX_LAYOUT_LINES) {
121                 input_lines[input_line_count++] = {i + 1, i + 1};
122             }
123             current_line = "";
124             continue;
125         }
126
127         String candidate = current_line;
128         candidate += value;
129         if (current_line.length() > 0 &&
130             canvas.textWidth(candidate) > text_width) {
131             input_lines[input_line_count - 1].end = i;
132             if (input_line_count < MAX_LAYOUT_LINES) {
133                 input_lines[input_line_count++] = {i, i};
134             }
135             current_line = "";
136         }
137
138         current_line += value;
139         input_lines[input_line_count - 1].end = i + 1;
140     }
141 }
142
143 uint16_t cursorLine()
144 {
145     // Find the visual line containing the current insertion point.
146     uint16_t cursor_line = 0;
147     for (uint16_t i = 0; i < input_line_count; ++i) {
148         if (cursor_position < input_lines[i].start) {
149             break;
150         }
151         cursor_line = i;
152         if (cursor_position < input_lines[i].end ||
153             cursor_position == input_lines[i].start) {
154             break;
155         }
156     }
157     return cursor_line;
158 }
159
160 void moveCursorVertical(int8_t direction)
161 {
162     // Preserve the column when moving between visual lines.
163     buildInputLayout();
164
165     const uint16_t current_line = cursorLine();

```

```

166     int16_t target_line = static_cast<int16_t>(current_line) + direction;
167     if (target_line < 0 ||
168         target_line >= static_cast<int16_t>(input_line_count)) {
169         return;
170     }
171
172     const uint16_t column = cursor_position -
173         input_lines[current_line].start;
174     const uint16_t target_length =
175         input_lines[target_line].end - input_lines[target_line].start;
176     cursor_position = input_lines[target_line].start +
177         (column < target_length ? column : target_length);
178 }
179
180 void moveCursorToLineEdge(bool end)
181 {
182     // Move to the beginning or end of the current visual line.
183     buildInputLayout();
184     const uint16_t current_line = cursorLine();
185     cursor_position = end ? input_lines[current_line].end
186         : input_lines[current_line].start;
187 }
188
189 void logSpecialKey(const char* name)
190 {
191     last_key = name;
192     Serial.printf("Character: [%s]\r\n", name);
193 }
194
195 enum NavAction : uint8_t {
196     NAV_LOG, NAV_HOME, NAV_PAGE_UP, NAV_DELETE, NAV_END, NAV_PAGE_DOWN,
197     NAV_RIGHT, NAV_LEFT, NAV_DOWN, NAV_UP
198 };
199
200 struct NavKey {
201     uint8_t key;
202     const char* name;
203     NavAction action;
204 };
205
206 static const NavKey nav_keys[] = {
207     // Map navigation usages, including the numeric keypad alternatives.
208     {0x49, "INSERT", NAV_LOG}, {0x4A, "HOME", NAV_HOME},
209     {0x4B, "PAGE UP", NAV_PAGE_UP}, {0x4C, "DELETE", NAV_DELETE},
210     {0x4D, "END", NAV_END}, {0x4E, "PAGE DOWN", NAV_PAGE_DOWN},
211     {0x4F, "ARROW RIGHT", NAV_RIGHT}, {0x50, "ARROW LEFT", NAV_LEFT},
212     {0x51, "ARROW DOWN", NAV_DOWN}, {0x52, "ARROW UP", NAV_UP},
213     {0x59, "KEYPAD END", NAV_END}, {0x5A, "KEYPAD DOWN", NAV_DOWN},
214     {0x5B, "KEYPAD PAGE DOWN", NAV_PAGE_DOWN},
215     {0x5C, "KEYPAD LEFT", NAV_LEFT}, {0x5E, "KEYPAD RIGHT", NAV_RIGHT},
216     {0x5F, "KEYPAD HOME", NAV_HOME}, {0x60, "KEYPAD UP", NAV_UP},
217     {0x61, "KEYPAD PAGE UP", NAV_PAGE_UP},
218     {0x62, "KEYPAD INSERT", NAV_LOG},
219     {0x63, "KEYPAD DELETE", NAV_DELETE}
220 };

```

```

221
222 bool handleNavigationKey(uint8_t key)
223 {
224     // NumLock changes keypad navigation keys into numeric input.
225     if (key >= 0x59 && key <= 0x63 && num_lock) {
226         return false;
227     }
228
229     const NavKey* nav = nullptr;
230     for (const NavKey& item : nav_keys) {
231         if (item.key == key) {
232             nav = &item;
233             break;
234         }
235     }
236     if (nav == nullptr) return false;
237
238     switch (nav->action) {
239     case NAV_HOME: moveCursorToLineEdge(false); break;
240     case NAV_PAGE_UP: moveCursorVertical(-INPUT_MAX_LINES); break;
241     case NAV_DELETE:
242         if (cursor_position < typed_text.length()) {
243             typed_text.remove(cursor_position, 1);
244         }
245         break;
246     case NAV_END: moveCursorToLineEdge(true); break;
247     case NAV_PAGE_DOWN: moveCursorVertical(INPUT_MAX_LINES); break;
248     case NAV_RIGHT:
249         if (cursor_position < typed_text.length()) ++cursor_position;
250         break;
251     case NAV_LEFT:
252         if (cursor_position > 0) --cursor_position;
253         break;
254     case NAV_DOWN: moveCursorVertical(1); break;
255     case NAV_UP: moveCursorVertical(-1); break;
256     default: break;
257     }
258     logSpecialKey(nav->name);
259     return true;
260 }
261
262 void renderTypedText()
263 {
264     // Draw the editor, status indicators, and cursor in one frame.
265     limitTextLength();
266
267     canvas.fillSprite(BLACK);
268     canvas.setTextColor(WHITE, BLACK);
269     canvas.setTextSize(2);
270     canvas.setTextWrap(false, false);
271
272     canvas.setCursor(8, 5);
273     canvas.print("USB KEYBOARD");
274     canvas.setCursor(248, 5);
275     canvas.print(keyboard_connected ? "READY" : "WAIT");

```

```

276 canvas.drawLine(0, 28, 319, 28, WHITE);
277
278 canvas.drawRect(0, 34, 319, 204, WHITE);
279
280 // Keep wrapping inside the input box and keep the cursor visible.
281 buildInputLayout();
282 const uint16_t cursor_line = cursorLine();
283
284 uint16_t first_line = 0;
285 if (input_line_count > INPUT_MAX_LINES) {
286     first_line = input_line_count - INPUT_MAX_LINES;
287     if (cursor_line < first_line) {
288         first_line = cursor_line;
289     } else if (cursor_line >= first_line + INPUT_MAX_LINES) {
290         first_line = cursor_line - INPUT_MAX_LINES + 1;
291     }
292 }
293
294 canvas.setClipRect(4, 38, 312, 165);
295 for (uint16_t i = first_line;
296     i < input_line_count && i < first_line + INPUT_MAX_LINES; ++i) {
297     canvas.setCursor(INPUT_LEFT,
298                     INPUT_TOP + (i - first_line) * INPUT_LINE_HEIGHT);
299     canvas.print(typed_text.substring(input_lines[i].start,
300                                     input_lines[i].end));
301 }
302
303 const String cursor_prefix =
304     typed_text.substring(input_lines[cursor_line].start, cursor_position);
305 canvas.setCursor(
306     INPUT_LEFT + canvas.textWidth(cursor_prefix),
307     INPUT_TOP + (cursor_line - first_line) * INPUT_LINE_HEIGHT);
308 canvas.print("_");
309 canvas.clearClipRect();
310
311 canvas.drawLine(0, 209, 319, 209, WHITE);
312 canvas.setTextWrap(false, false);
313 canvas.setTextSize(1.25f);
314 canvas.setCursor(8, 212);
315 canvas.printf("CAPS %s", caps_lock ? "ON" : "OFF");
316 canvas.setCursor(160, 212);
317 canvas.printf("NUM %s", num_lock ? "ON" : "OFF");
318 canvas.setCursor(8, 226);
319 canvas.printf("LEN %u", static_cast<unsigned>(typed_text.length()));
320 canvas.setCursor(80, 226);
321 canvas.printf("LAST %s", last_key.c_str());
322
323 canvas.pushSprite(0, 0);
324 }
325
326 void appendKeyboardReport(uint8_t const* report, uint16_t len)
327 {
328     // Parse the standard 8-byte boot keyboard report.
329     // Byte 0 holds modifiers; bytes 2-7 hold up to six key usages.
330     if (len < 8) {

```

```

331     return;
332 }
333
334 // Left/right Shift bits are 0x02 and 0x20 in the modifier byte.
335 const bool shifted = (report[0] & 0x22) != 0;
336 for (uint8_t i = 2; i < 8; ++i) {
337     const uint8_t key = report[i];
338     if (key == 0 || !isNewKey(key)) {
339         continue;
340     }
341
342     // Toggle lock keys, handle navigation, then insert printable text.
343     if (key == 0x39) {
344         caps_lock = !caps_lock;
345         last_key = caps_lock ? "CAPS ON" : "CAPS OFF";
346         Serial.printf("Character: [CAPS LOCK %s]\r\n",
347             caps_lock ? "ON" : "OFF");
348     } else if (key == 0x53) {
349         num_lock = !num_lock;
350         last_key = num_lock ? "NUM ON" : "NUM OFF";
351         Serial.printf("Character: [NUM LOCK %s]\r\n",
352             num_lock ? "ON" : "OFF");
353     } else if (handleNavigationKey(key)) {
354     } else if (key == 0x28 || key == 0x58) {
355         insertText("\n");
356         logSpecialKey("ENTER");
357     } else if (key == 0x2A) {
358         if (cursor_position > 0) {
359             typed_text.remove(cursor_position - 1, 1);
360             --cursor_position;
361         }
362         logSpecialKey("BACKSPACE");
363     } else if (key == 0x2B) {
364         insertText(" ");
365         logSpecialKey("TAB");
366     } else {
367         char value = keyToAscii(key, shifted);
368         if (value == 0) {
369             value = keypadToAscii(key);
370         }
371         if (value != 0) {
372             String insertion;
373             insertion += value;
374             insertText(insertion);
375             last_key = String(value);
376             Serial.printf("Character: '%c' (HID usage: 0x%02X)\r\n",
377                 value, key);
378         }
379     }
380 }
381
382 // Save this report so held keys are ignored on the next report.
383 for (uint8_t i = 0; i < 6; ++i) {
384     previous_keys[i] = report[i + 2];
385 }

```

```

386     renderTypedText();
387 }
388
389 void setup()
390 {
391     // Initialize the display, editor state, serial port, and USB host.
392     M5.begin();
393     Serial.begin(115200);
394
395     canvas.createSprite(320, 240);
396     typed_text = "";
397     last_key = "START";
398     num_lock = false;
399     keyboard_connected = false;
400     cursor_position = 0;
401     renderTypedText();
402
403     Serial.println("M5Module USB HOST Keyboard");
404     if (USBHost.begin(0)) {
405         Serial.println("MODULE INIT OK");
406         last_key = "HOST READY";
407         renderTypedText();
408     } else {
409         Serial.println("MODULE INIT ERROR");
410         last_key = "HOST ERROR";
411         renderTypedText();
412     }
413 }
414
415 void loop()
416 {
417     // Poll the USB host so keyboard reports reach the HID callback.
418     USBHost.task();
419     delay(1);
420 }
421
422 void tuh_hid_mount_cb(uint8_t dev_addr, uint8_t instance,
423                      uint8_t const* desc_report, uint16_t desc_len)
424 {
425     // Start requesting reports after a keyboard is connected.
426     (void)desc_report;
427     (void)desc_len;
428
429     uint16_t vid;
430     uint16_t pid;
431     tuh_vid_pid_get(dev_addr, &vid, &pid);
432     Serial.printf("HID mounted: address=%d, instance=%d, VID=%04X, PID=%04X\r\n",
433                  dev_addr, instance, vid, pid);
434     keyboard_connected = true;
435     last_key = "CONNECTED";
436     renderTypedText();
437
438     if (!tuh_hid_receive_report(dev_addr, instance)) {
439         Serial.println("Cannot request HID report");
440     }

```

```

441 }
442
443 void tuh_hid_umount_cb(uint8_t dev_addr, uint8_t instance)
444 {
445     // Update the screen when the keyboard is disconnected.
446     Serial.printf("HID unmounted: address=%d, instance=%d\r\n",
447                 dev_addr, instance);
448     keyboard_connected = false;
449     last_key = "DISCONNECTED";
450     renderTypedText();
451 }
452
453 void tuh_hid_report_received_cb(uint8_t dev_addr, uint8_t instance,
454                                uint8_t const* report, uint16_t len)
455 {
456     // Print both the raw report and the decoded key information.
457     Serial.print("Keyboard report: ");
458     for (uint16_t i = 0; i < len; ++i) {
459         Serial.printf("0x%02X ", report[i]);
460     }
461     Serial.println();
462
463     appendKeyboardReport(report, len);
464
465     if (len >= 8) {
466         Serial.printf("Modifiers: 0x%02X, Keys:", report[0]);
467         for (uint8_t i = 2; i < 8; ++i) {
468             if (report[i] != 0) {
469                 Serial.printf(" 0x%02X", report[i]);
470             }
471         }
472         Serial.println();
473     }
474
475     if (!tuh_hid_receive_report(dev_addr, instance)) {
476         Serial.println("Cannot request next HID report");
477     }
478 }

```

After the program starts and the module initializes successfully, the CoreS3 screen shows the USB host and keyboard status, and the serial monitor outputs information similar to the following:

```

M5Module USB HOST Keyboard
MODULE INIT OK
HID mounted: address=1, instance=0, VID=XXXX, PID=XXXX
Keyboard report: 0x00 0x00 0x04 0x00 0x00 0x00 0x00 0x00
Character: 'a' (HID usage: 0x04)
Modifiers: 0x00, Keys: 0x04

```

When you enter characters or use editing and navigation keys, the text and cursor on the CoreS3 screen update together. The bottom shows keyboard status and the most recent key, while the serial port outputs HID reports and decoded characters.



4. Compile and Upload

- 1. Enter download mode: Press and hold the CoreS3 reset button for about 2 seconds until the internal green LED turns on, then release the button. The green LED turns off after release, indicating that the device has entered download mode and is ready for upload.

Note

Devices must enter download mode before uploading. This step may differ between host devices. See the device upload tutorial list at the bottom of the [Arduino IDE guide](#) for specific instructions.



- 2. Select the device port, click the Compile and Upload button in the upper-left corner of Arduino IDE, and wait for the program to compile and upload.

