

920MHz Private LoRa 多機能開発ボード (DTP-DBLR-22SR2) サンプル / 基板検査プログラム

概要

本ソフトウェア **DevBord_E220** は、**920MHz Private LoRa 多機能開発ボード DevBoard DTP-DBLR-22SR2** の、動作 および 基板検査 (TestModule、LoRa送信動作を含む) を行う PlatformIO / Arduino サンプル・プログラムです。

起動時のスイッチ状態によって、次の動作を行います。

- **SW3を押した状態(ON)で起動**: 基板検査モード
- **通常起動 + SW6 OFF**: 受信側 Receiver として動作
- **通常起動 + SW6 ON**: 送信側 Sender として動作

送信側では、実装されている GPS・SHT30・ADXL345 のデータを取得し、LoRa で送信します。

受信側では、受信データを UART0():USB-C に接続されたPCへ送信し、LCDに受信時のRSSIを表示します。

1. 開発環境

- PlatformIO
- Framework: Arduino
- Board: **esp32-s3-devkitc-1**
- Serial Monitor: 115200 bps
- ESP32-S3 内蔵 USB CDC を使用

platformio.ini では、書き込み・モニタ用ポートとして **COM14** が指定されています。

```
[env:esp32-s3-devkitc-1]
platform = espressif32
board = esp32-s3-devkitc-1
framework = arduino

upload_port = COM14
monitor_port = COM14
monitor_speed = 115200

build_flags =
    -DARDUINO_USB_MODE=1
    -DARDUINO_USB_CDC_ON_BOOT=1
```

使用するPCに合わせて COM ポートは変更してください。

2. 主なハードウェア

デバイス	用途	接続
ESP32-S3	メインCPU	-
E220-900T22S(JP)	LoRa通信	UART2
SHT30	温湿度センサ	I2C
AQM0802A	8桁×2行 LCD	I2C
ADXL345	3軸加速度センサ	SPI
microSD	SDカード	SPI
E108-GN02	GPS	UART1
CP2102N	Receiver時のPC向けシリアル出力	UART0
D4/D5/D6	動作確認LED	GPIO

3. プロジェクト構成

```
DevBoard_E220/  
├─ platformio.ini  
├─ README.md  
└─ src/  
    ├─ main.cpp  
    ├─ modules.cpp  
    ├─ modules.h  
    ├─ testModules.cpp  
    ├─ globals.cpp  
    ├─ globals.h  
    ├─ ZigBee.h  
    │  
    └─ --- E220ライブラリ ---  
        ├─ firmware.h  
        ├─ esp32_e220900t22s_jp_lib_v2.cpp  
        └─ esp32_e220900t22s_jp_lib_v2.h
```

main.cpp

アプリケーションのメイン処理です。

- GPIO初期化
- SW3による基板検査モード選択
- SW6によるLoRa送信／受信モード選択
- LoRa初期化
- センサ存在確認
- LoRa受信タスク
- LoRa送信タスク
- GPS / SHT30 / ADXL345 データ取得 ～ LoRa送信

modules.cpp / modules.h

基板上の各デバイスを制御します。

- GPIO初期化
- LED制御
- LCD制御
- SHT30
- ADXL345
- GPS
- microSD確認
- LoRa初期化
- SW7状態確認
- 実装モジュール検出

testModules.cpp

基板検査用の処理です。(4. 基板検査モード を参照)

globals.cpp / globals.h

アプリケーション全体で使用するグローバル変数を定義します。

主なフラグ：

```
LCD_Exist  
SHT30_Exist  
ADXL345_Exist  
SD_Exist  
GPS_Exist  
LoRaSender
```

esp32_e220900t22s_jp_lib_v2.cpp / .h

E220-900T22S(JP) 用LoRaライブラリです。

- LoRa設定ファイル読み込み
- E220レジスタ設定
- 送信
- 受信
- Normal / WOR / Configuration モード切替
- Strict Mode関連処理

firmware.h

E220 のファームウェアバージョン／動作モードを定義します。

現在は、

```
#define FIRMWARE_VERSION 0x20
```

となっており、22SR2 Strict Mode 用設定です。

4. 起動モード

起動時に SW3 と SW6 の状態を読み取ります。

SW3 ON（数秒間押した状態）で起動

基板検査モードになります。

```
if (digitalRead(PIN_SW3) == LOW) {  
    testModules();  
    while(1) { delay(1); }  
}
```

検査終了後は通常動作には移行せず停止します。

SW3 OFFで起動

通常動作になります。

起動時の SW6 の状態で送信側／受信側を決定します。

シリアルモニタへ、

```
=== DevBoard DTP-DBLR-22SR2 Execution Start ===
```

と表示します。

SW6

```
LoRaSender = false;  
  
if (digitalRead(PIN_SW6) == LOW)  
    LoRaSender = true;
```

したがって、

SW6	動作
LOW / ON	LoRa Sender
HIGH / OFF	LoRa Receiver

となります。

Sender時 LCD に " LoRaMode Sender " 表示

シリアルモニタへ " === RUN as Sender === " と表示

Receiver時 LCD に " LoRaMode Receiver " 表示

シリアルモニタへ " === RUN as Receiver === " と表示

5. 部品名称 と 主なGPIO割り当て

I2C

信号	GPIO
SDA	8
SCL	9

SHT30 と AQM0802A LCD が同じI2Cバスを使用します。

SPI

信号	GPIO
SCLK	14
MISO	21
MOSI	47
ADXL345 CS	13
microSD CS	48

ADXL345 と microSD はSPIバスを共用しています。

LED

LED	GPIO
D4	10
D5	11
D6	12

HIGHで点灯します。

モーメンタリスイッチ

SW	GPIO
SW3	4
SW4	5
SW5	6
SW6	7

押下時 LOW です。

SW3 は基板検査時の ENTER として使
います。

DIPスイッチ (SW7)

SW	GPIO
E220_M0	1
E220_M1	2

SW7 は E220-900T22S(R2) の M0,M1 PIN
として使用します。

R24/R25(1K) 経由で、R23/R33(10K) で
3V3 プルアップされ、ON にすると GND
に落ちます。

UART0 / CP2102N / USB-C

Receiver動作時は ESP32-S3 の UART0 を
CP2102N に接続し、2個目のUSB-Cポート
からPCへLoRa受信情報を出力します。

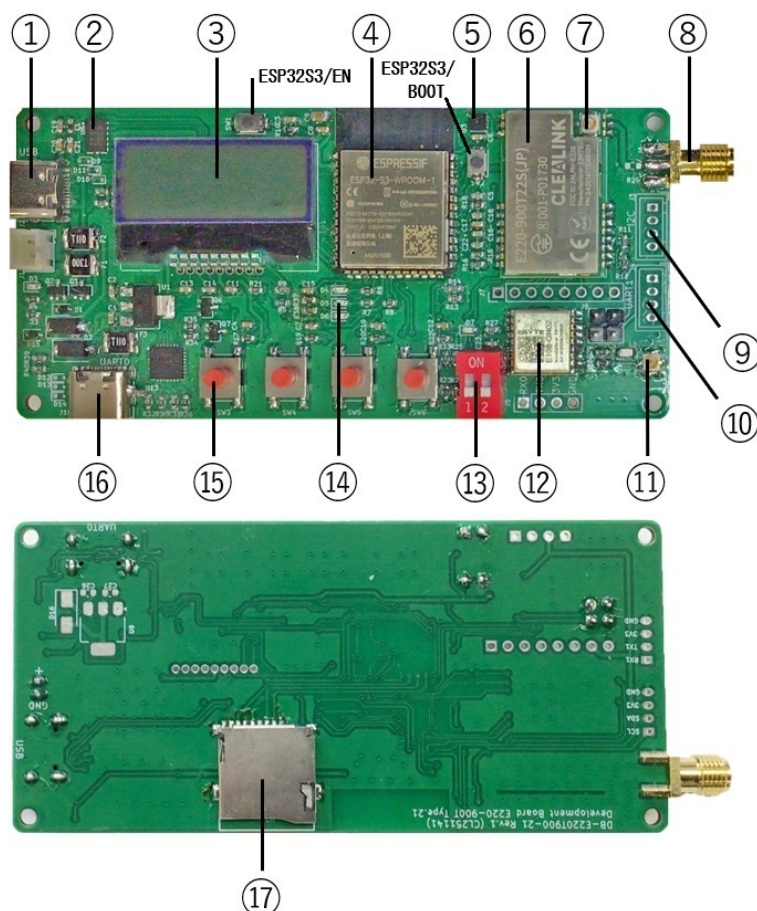
信号	GPIO
UART0 RX	44
UART0 TX	43

初期化：

```
SerialCP2102.begin(115200,
SERIAL_8N1, 44, 43);
```

USBポートの役割は次のとおりです。

USB-C 1 : ESP32-S3 Native
USB



1. USB type-c (USB信号直結)
2. 加速度センサー (ADXL345)
3. 小型液晶モニター (AQM0802A)
4. ESP32-S3-WROOM-1
5. 温湿度センサー (SHT30)
6. LoRa通信モジュール (E220-900T22S (JP) R2)
7. U. FL (IPEX) アンテナ端子 (LoRa用)
8. SMAアンテナコネクタ (LoRa用)
9. I2Cコネクタ (Groveコネクタ互換 HY 4P タイプ)
10. UARTコネクタ (Groveコネクタ互換 HY 4P タイプ)
11. U. FL (IPEX) アンテナ端子 (GNSS用)
12. GNSSモジュール (E108-GN02)
13. DIPスイッチ (LoRaモジュール M0、M1 制御用)
14. LED ×3
15. モーメンタリスイッチ ×4
16. USB type-c (シリアル変換ICを通して、ESP32のUARTへ接続)
17. microSDカードスロット

```

        -> Serial
        -> 書込み / デバッ
グ

USB-C 2 : ESP32-S3 UART0
        -> CP2102N
        -> SerialCP2102
        -> Receiver受信デ
ータ出力

```

`SerialCP2102` の実体は `modules.cpp` に、

```

HardwareSerial
SerialCP2102(0);

```

として定義し、`modules.h` では、

```

extern HardwareSerial
SerialCP2102;

```

として参照します。

E220-900T22S(JP)

信号	GPIO
M0	1
M1	2
AUX	3
ESP32 TX → E220 RX	15
ESP32 RX ← E220 TX	16

GPS E108-GN02

信号	GPIO
ESP32 TX → GPS RX	17
ESP32 RX ← GPS TX	18
RESET	41
FORCE_ON	42

GPS UART速度は 9600 bps です。

6. 基板検査モード

SW3を押しながら起動すると、

```
=== DevBoard DTP-DBLR-22SR2 Test Start ===
```

と表示され、`testModules()` が開始されます。

検査は概ね以下の順番です。

```
LCD
↓
LED
↓
Button
↓
DIP SW
↓
I2C
↓
SHT30
↓
ADXL345
↓
microSD
↓
LoRa Config
↓
LoRa Send
↓
GPS
↓
Test Finish
```

原則としてSW3（ENTER）で次の項目へ進みます。

6.1 LCD

AQM0802AをI2Cアドレス `0x3E` で確認します。

検出できた場合はコントラストを設定し、`Hello!` を表示します。

LCDはオプション扱いのため、未実装でも検査自体は継続します。

6.2 LED

D4 / D5 / D6を順番に点灯した後、3個同時点灯を行います。

目視で点灯状態を確認します。

6.3 モーメンタリスイッチ

SW3～SW6の状態をSerialとLCDへ表示します。

SW3を押すとボタン検査を終了します。

6.4 I2Cスキャン

I2Cアドレス **0x01** ～ **0x7E** を走査します。

既知のデバイスの場合は次のように識別します。

```
0x3E : AQM0802A
0x44 : SHT30-DIS
```

6.5 SHT30

単発測定で温度・湿度を取得します。

- CRC-8チェックあり
- 温度: degC
- 湿度: %RH
- ALERT: GPIO38

6.6 ADXL345

4線式SPIで通信します。

主な設定:

- SPI MODE3
- 4 MHz
- FULL_RES
- ±16 g
- 100 Hz
- 3.9 mg/LSB

DEVID = **0xE5** を確認した後、X/Y/Zの生値とg値を表示します。

6.7 microSD

microSDをマウントし、カード容量を表示します。

ADXL345とSPIバスを共有しているため、SDアクセス時はADXL345のCSをHIGHにします。

検査項目間は ENTER (SW3) で進みます。

6.8 LoRa検査

E220をConfiguration Modeに切り替えてレジスタを読み出します。

確認内容には、

- 自局アドレス
- UART速度
- 無線速度
- チャンネル
- 固定／透過送信
- RSSI Byte
- Firmware Version

などがあります。

設定読出し成功後、**hello, world!** をLoRa送信します。

7. 通常起動時の処理

概略は以下です。

```
Power ON
|
v
initModulesGPIO()
|
v
Serial.begin()
|
+---- SW3 ON ----> testModules() ---> 停止
|
v
通常動作
|
v
起動LED表示
|
v
SW6確認
|
v
SW7確認
|
v
initLoRa()
|
+---- Sender ----> checkModules()
|
v
LoRaRecvTask 起動
```

```
|  
v  
LoRaSendTask 起動  
|  
v  
loop()
```

7.1 SW7確認について

SW7 は E220 の M0/M1 に接続されています。

通常動作開始時には、

```
checkLoRaSW7();
```

でSW7を確認します。

SW7がONの場合は E220 のモードをプログラムから正常に切り替えられないため、

```
SW7 が ON のためモード切替ができません。
```

と表示し、SW7がOFFになるまで待ちます。

7.2 LoRa初期化

`initLoRa()` で以下を実行します。

1. E220 UARTを終了
2. 9600 bpsで再オープン
3. E220をConfiguration Modeへ変更
4. LoRa設定ファイルをSPIFFSから読み込み
5. E220レジスタを設定
6. Normal Modeへ移行

LoRa設定ファイルが存在しない場合は、ライブラリ内部のデフォルト設定値が使用されます。

現在の `FIRMWARE_VERSION` は `0x20` のため、設定ファイル名は、

```
/e220900t22s_jp_config_v20.ini
```

です。

設定ファイルを読み込めない場合の主なデフォルト値は以下です。

項目	デフォルト
Own Address	0x0000
UART	9600 bps
Air Data Rate	SF9 / BW125
Sub Packet	200 bytes
TX Power	13 dBm
Channel	0
RSSI Byte	Disable
Transmission	Fixed
WOR Cycle	2000 ms
Encryption Key	0x0000
Target Address	0x0000
Target Channel	0
Strict Mode	Enable

7.3 Sender動作

SW6をONにして起動すると Sender になります。

最初に、

```
checkModules();
```

を実行し、

- LCD
- SHT30
- ADXL345
- microSD
- GPS

の実装状態を確認します。

その後 `loop()` で存在するセンサのデータだけを取得します。

```
if (GPS_Exist) {  
    GPS_Read(buf);  
    strcat(SendData, buf);  
}
```

```
if (SHT30_Exist) {  
    SHT30_Read(buf);  
    strcat(SendData, buf);  
}  
  
if (ADXL345_Exist) {  
    ADXL345_Read(buf);  
    strcat(SendData, buf);  
}
```

データ作成後、

```
SendRequest = 1;
```

として `LoRaSendTask` に送信を依頼します。

現在は約8秒周期です。

```
delay(8000);
```

7.3.1 SHT30

SHT30から、

- 温度
- 湿度
- ALERT端子状態

を取得し、LoRa送信データ `SendData` に `strcat` します。

送信用文字列の例：

```
Temp: 25.30 degC  
Humd: 48.20 %RH  
ALERT(IO38): HIGH
```

CRC-8チェックも実施しています。

7.3.2 ADXL345

ADXL345は4線式SPIで接続されています。

設定は、

- FULL_RES
- $\pm 16g$
- 100 Hz
- Measure Mode

です。

取得データはRaw値とg値へ変換して使用します。

送信用文字列の例：

```
raw:    10    -20    256  |  g: +0.039 -0.078 +0.998
```

LoRa送信データ SendData に strcat します。

7.3.3 GPS

GPSには E108-GN02 を使用します。

UART設定：

```
9600 bps / 8N1
```

`GPS_ReadLine()` は \$ から始まるNMEAセンテンスを1行取得します。

送信用文字列の例：

```
$GNRMC,001339.042,V,0000.00000,N,00000.00000,E,0.000,0.00,060180,,N*59
```

取得後、LoRa送信データ SendData へ strcat して、LoRa送信します。

7.3.4 LoRa送信タスク

`LoRaSendTask()` は `SendRequest` を監視します。

送信要求があると、

```
lora.SendFrame(  
    lora.config,  
    (uint8_t *)SendData,  
    SendDataLength  
);
```

を実行します。

送信中はLCDに、

```
Sending!
```

送信終了後は、

```
Wait...
```

を表示します。

送信後、

```
SendData[0] = '\0';  
SendRequest = 0;
```

として次の送信を待ちます。

7.4 Receiver動作 (LoRa受信タスク)

LoRaRecvTask() は常時、

```
lora.ReceiveFrame(&lora.data)
```

を実行してLoRa受信を監視します。

Receiver時には UART0 / CP2102N を115200 bpsで初期化し、LoRa受信結果を2個目のUSB-Cポートへ出力します。

```
SerialCP2102.begin(115200, SERIAL_8N1, 44, 43);
```

受信すると SerialCP2102 へ、

- 受信データ (HEX)
- RSSI

を出力します。

```
E220  
-> ESP32-S3
```

```
-> UART0  
-> CP2102N  
-> USB-C  
-> Windows COMポート
```

現在のHEX表示は最大約33バイトで打ち切る処理になっています。Native USB側の **Serial** はデバッグ用として独立して使用できます。

8. microSD LOGファイル

Sender動作時にmicroSDが存在する場合、起動時に新しいLOGファイルを1個作成します。

ファイル名はNVS（Flash）に保存したシーケンス番号から生成します。

```
/LOG00000.TXT  
/LOG00001.TXT  
/LOG00002.TXT  
...
```

ファイル名の形式は、

```
"/LOG%05lu.TXT"
```

です。

8.1 起動時のLOGファイル作成

`createLogFile()` は次の順序で処理します。

```
NVSから sequence を読出し  
    |  
    v  
LOGxxxxx.TXT を作成  
    |  
    v  
"LOG Sequence = xxxxx" を書き込み  
    |  
    v  
close()  
    |  
    v  
sequence++  
    |  
    v  
次回用sequenceをNVSへ保存
```

NVSでは名前空間 `logger`、キー `sequence` を使用します。

```
prefs.begin("logger", true);
uint32_t seq = prefs.getUInt("sequence", 0);
prefs.end();
```

ファイル作成に成功した後、次回起動用の番号を保存します。

```
logSequence++;
saveLogSequence(logSequence);
```

したがって、電源をOFF/ONしてもシーケンス番号を継続できます。

8.2 LOGデータの追記

通常動作中は、LoRa送信用に作成した `SendData` を、

```
appendLogData(SendData);
```

で現在のLOGファイルへ追記します。送信用文字列の例：

```
$GNRMC,001501.042,V,0000.00000,N,00000.00000,E,0.000,0.00,060180,,N*54
Temp: 35.98 degC
Humd: 38.24 %RH
ALERT(IO38): HIGH
raw:      4      11      264 | g:  +0.016  +0.043  +1.030
```

`appendLogData()` は追記のたびに、

```
SD.open(FILE_APPEND)
  |
  v
file.print(data)
  |
  v
file.close()
```

を行います。

ファイルを開きっぱなしにしないため、各書込み終了時点でmicroSDへファイルを閉じる構成です。

8.3 LOG関連の変数

変数	型	役割
logSequence	uint32_t	LOGファイルのシーケンス番号
logFileName	char[]	現在使用中のLOGファイル名
prefs	Preferences	NVSのシーケンス番号読書き用

9. FreeRTOSタスク

通常動作では2つのタスクを作成しています。

```
xTaskCreateUniversal(
    LoRaRecvTask,
    "LoRaRecvTask",
    8192,
    NULL,
    1,
    NULL,
    APP_CPU_NUM
);

xTaskCreateUniversal(
    LoRaSendTask,
    "LoRaSendTask",
    8192,
    NULL,
    1,
    NULL,
    APP_CPU_NUM
);
```

役割は、

```
loop()
|
+-- センサ/GPSデータ作成
|
+-- SendRequest = 1
    |
    v
    LoRaSendTask
    |
    v
    LoRa送信

LoRaRecvTask
```

```
|  
+-- 常時LoRa受信監視
```

となっています。

10. ビルド

PlatformIOから通常通りビルドします。

VS Codeの場合：

```
PlatformIO: Build
```

またはターミナルから、

```
pio run
```

書き込みは、

```
pio run -t upload
```

です。

11. シリアルモニタ

通信速度は、

```
115200 bps
```

です。

PlatformIOでは、

```
pio device monitor
```

で確認できます。

12. 補足

- LCDはオプション部品として扱われており、未実装でも処理を継続します。
- ADXL345とmicroSDは同じSPIバスを共用するため、使用しない側のCSをHIGHにして競合を防止しています。
- SW7がONのままだとE220のM0/M1制御と競合するため、通常起動時はSW7をOFFにする必要があります。
- LoRa設定ファイルがSPIFFSに存在しない場合はデフォルト設定が使用されます。
- Sender側では実装されているモジュールだけを自動検出して送信データに含めます。

13. 主な動作まとめ



14. 関数一覧

この章では、ソースを追いやすように主要関数を **定義ファイル・引数・戻り値・役割** とともにまとめます。

14.1 main.cpp

関数	引数	戻り値	役割
setup()	なし	void	GPIO、Serial、起動モード、LoRa、モジュール確認、FreeRTOSタスクを初期化する
loop()	なし	void	Sender時にGPS/SHT30/ADXL345のデータを作成し、LoRa送信要求を発行する
LoRaRecvTask(void *pvParameters)	FreeRTOSタスク引数	void	LoRa受信を常時監視し、受信データとRSSIをSerialへ表示する
LoRaSendTask(void *pvParameters)	FreeRTOSタスク引数	void	SendRequest を監視し、送信要求時に CLoRa::SendFrame() を実行する

Receiver用シリアルオブジェクト

`SerialCP2102` はUART0用の `HardwareSerial` オブジェクトです。

```
HardwareSerial SerialCP2102(0);
```

Receiver時のみ `setup()` で115200 bpsに初期化し、LoRa受信データとRSSIをPCへ出力します。

14.2 modules.cpp

関数	引数	戻り値	役割
<code>initModulesGPIO()</code>	なし	<code>void</code>	LED、SW、LoRa、GPS、SPI関連GPIOを初期化する
<code>threetimesBlinkLED()</code>	なし	<code>void</code>	起動確認用にLEDを3回点滅させる
<code>startBlinkLED()</code>	なし	<code>void</code>	起動時のLED点滅処理を行う
<code>checkLoRaSW7()</code>	なし	<code>void</code>	E220のM0/M1に接続されたSW7状態を確認し、必要ならOFFになるまで待つ
<code>initLoRa()</code>	なし	<code>void</code>	E220 UART、設定読み込み、レジスタ設定、Normal Mode移行を行う
<code>sht30Crc(const uint8_t *data, size_t len)</code>	データ、長さ	<code>uint8_t</code>	SHT30受信データ用CRC-8を計算する内部関数
<code>sht30Read(float &tempC, float &humidity)</code>	温度・湿度の格納先	<code>bool</code>	SHT30から温湿度を読み、CRC確認後に値へ変換する
<code>SHT30_Read(char *buf)</code>	出力文字列バッファ	<code>void</code>	SHT30の温度・湿度・ALERT状態を送信用文字列へ整形する
<code>GPS_ReadLine(char *ret_str, unsigned int buf_size, uint32_t timeoutMs)</code>	出力バッファ、サイズ、タイムアウト	<code>int</code>	GPSから\$で始まるNMEAセンテンスを1行読み込む
<code>GetGNRMC(const char *buf)</code>	NMEA文字列	<code>bool</code>	受信NMEAがGNRMC/RMCとして利用可能か確認する
<code>GPS_Read(char *gpsbuf)</code>	出力文字列バッファ	<code>void</code>	GPSから必要なNMEAデータを取得し、送信用データへ格納する
<code>adxlWriteReg(uint8_t reg, uint8_t value)</code>	レジスタ、値	<code>void</code>	ADXL345レジスタへSPIで1バイト書き込む内部関数

関数	引数	戻り値	役割
<code>adxlReadReg(uint8_t reg)</code>	レジスタ	<code>uint8_t</code>	ADXL345レジスタをSPIで1バイト読み込む
<code>adxlReadXYZ(int16_t &x, int16_t &y, int16_t &z)</code>	X/Y/Z格納先	<code>void</code>	ADXL345の3軸Rawデータを一括読み込みする
<code>adxlBegin()</code>	なし	<code>bool</code>	ADXL345のDEVID確認と測定条件設定を行う
<code>ADXL345_Read(char *buf)</code>	出力文字列バッファ	<code>void</code>	ADXL345のRaw値とg値を読み、文字列へ整形する
<code>loadLogSequence()</code>	なし	<code>uint32_t</code>	NVSの <code>logger/sequence</code> から次に使用するLOG番号を読み出す内部関数
<code>saveLogSequence(uint32_t seq)</code>	シーケンス番号	<code>void</code>	次回起動用LOG番号をNVSへ保存する内部関数
<code>createLogFile()</code>	なし	<code>bool</code>	NVSの番号で新しいLOGファイルを作成し、成功後に次回番号をNVSへ保存する
<code>appendLogData(const char *data)</code>	追記文字列	<code>bool</code>	現在のLOGファイルを追記モードで開き、データを書いて閉じる
<code>checkModules()</code>	なし	<code>void</code>	LCD、SHT30、ADXL345、microSD、GPSの実装有無を確認してフラグへ反映する

`sht30Crc()`、`adxlWriteReg()`、`loadLogSequence()`、`saveLogSequence()` は `modules.cpp` 内だけで使用する `static` 内部関数です。

14.3 testModules.cpp

関数	引数	戻り値	役割
<code>e220WaitAux(bool level, uint32_t timeoutMs)</code>	期待AUXレベル、タイムアウト	<code>bool</code>	E220 AUX端子が指定状態になるまで待つ
<code>e220SetMode(E220Mode mode)</code>	E220動作モード	<code>void</code>	M0/M1を制御してE220の動作モードを変更する
<code>e220ReadDipState(bool &m0Pulled, bool &m1Pulled)</code>	M0/M1状態格納先	<code>void</code>	SW7によるM0/M1のDIP状態を読み取る
<code>e220ReadRegs(uint8_t addr, uint8_t len, uint8_t *out)</code>	開始アドレス、長さ、	<code>bool</code>	E220設定レジスタを読み出す

関数	引数	戻り値	役割
出力先			
<code>e220BaudValue(uint8_t code)</code>	E220ボーレートコード	<code>uint32_t</code>	レジスタ値を実際のUARTボーレートへ変換する
<code>e220PrintConfig(const uint8_t *r)</code>	レジスタデータ	<code>void</code>	読み出したE220設定を人が読める形式でSerial表示する
<code>waitForButtonPressDebounce(uint8_t pin, unsigned long debounceMs)</code>	GPIO、デバウンス時間	<code>void</code>	チャタリングを除去しながらボタン押下を待つ
<code>buttonTestMode()</code>	なし	<code>void</code>	SW3～SW6の基板検査を行う
<code>dipSwitchTestMode()</code>	なし	<code>void</code>	SW7 DIPスイッチの基板検査を行う
<code>testModules()</code>	なし	<code>void</code>	LCD、LED、SW、センサ、SD、LoRa、GPSを順番に検査する基板検査メイン関数

`e220WaitAux()`、`e220SetMode()`、`e220ReadRegs()`、`e220BaudValue()`、`e220PrintConfig()`、`waitForButtonPressDebounce()`、`buttonTestMode()`、`dipSwitchTestMode()` は `testModules.cpp` 内部用の `static` 関数です。

14.4 SimpleLCD クラス関数

SimpleLCD は `modules.h` に定義された AQM0802A (ST7032i) 用の最小LCDドライバです。

メンバ関数	引数	戻り値	役割
<code>begin(uint8_t cols, uint8_t lines)</code>	桁数、行数	<code>bool</code>	LCD存在確認とST7032初期化を行う
<code>setContrast(uint8_t contrast)</code>	コントラスト値	<code>void</code>	LCDコントラストを設定する
<code>clear()</code>	なし	<code>void</code>	LCD表示をクリアする
<code>setCursor(uint8_t col, uint8_t row)</code>	桁、行	<code>void</code>	LCDカーソル位置を指定する
<code>print(const char *s)</code>	表示文字列	<code>void</code>	文字列をLCDへ表示する
<code>printf(const char *fmt, ...)</code>	printf形式	<code>void</code>	書式付き文字列を作成してLCDへ表示する

メンバ関数	引数	戻り値	役割
<code>command(uint8_t c)</code>	LCDコマンド	<code>bool</code>	ST7032へコマンドを送信する。クラス内部専用

グローバルインスタンスは、

```
extern SimpleLCD lcd;
```

として使用されます。

15. 主な関数呼び出し関係

通常起動時の代表的な関数呼び出しは次のようになります。

```

setup()
|
+-- initModulesGPIO()
|
+-- SW3 ON ?
|   |
|   +-- YES --> testModules()
|
+-- threetimesBlinkLED()
|
+-- checkLoRaSW7()
|
+-- initLoRa()
|   |
|   +-- CLoRa::SwitchToConfigurationMode()
|   +-- CLoRa::LoadConfigSetting()
|   +-- CLoRa::InitLoRaModule()
|   +-- CLoRa::SwitchToNormalMode()
|
+-- Sender ?
|   |
|   +-- YES --> checkModules()
|
+-- LoRaRecvTask()
+-- LoRaSendTask()

loop() [Sender時]
|
+-- GPS_Read()
|
+-- SHT30_Read()
|   +-- sht30Read()
```

```

|           +-- sht30Crc()
|
+-- ADXL345_Read()
|   +-- adxlReadXYZ()
|
+-- appendLogData()  [SD存在時]
|
+-- SendRequest = 1
|   |
|   v
|   LoRaSendTask()
|   |
|   +-- CLoRa::SendFrame()

LoRaRecvTask()
|
+-- CLoRa::ReceiveFrame()

```

基板検査側は、

```

testModules()
|
+-- LCD検査
+-- LED検査
+-- buttonTestMode()
+-- dipSwitchTestMode()
+-- I2C検査
+-- SHT30検査
+-- ADXL345検査
+-- microSD検査
+-- e220ReadRegs()
|   +-- e220PrintConfig()
|
+-- CLoRa::SendFrame()
|
+-- GPS検査

```

という構成です。

16. ソースを追うときの目安

アプリケーション動作を確認するときは、次の順番で読むと把握しやすくなります。

```

1. main.cpp
   ↓
2. modules.h
   ↓

```

```
3. modules.cpp
   ↓
4. testModules.cpp
   ↓
5. esp32_e220900t22s_jp_lib_v2.h
   ↓
6. esp32_e220900t22s_jp_lib_v2.cpp
```

特に通常動作だけを確認する場合は、

```
setup()
  ↓
initLoRa()
  ↓
loop()
  ↓
LoRaSendTask() / LoRaRecvTask()
```

を先に追い、その後に各センサ関数を見ると理解しやすくなります。

17. 付録 E220-900T22S(SR2) のライブラリ関数

詳細は、製品ページのダウンロードサポートからダウンロードしてご参照ください。

17.1 CLoRa クラス関数一覧

CLoRa は esp32_e220900t22s_jp_lib_v2.h / .cpp に実装された E220-900T22S(JP) 用ライブラリです。

アプリケーション本体から使用する代表的な関数は、

```
lora.LoadConfigSetting(...)
lora.InitLoRaModule(...)
lora.SwitchToConfigurationMode()
lora.SwitchToNormalMode()
lora.SendFrame(...)
lora.ReceiveFrame(...)
```

です。

以下にライブラリの関数をまとめます。

CLoRa は esp32_e220900t22s_jp_lib_v2.h / .cpp に実装された E220-900T22S(JP) 用ライブラリです。

17.2 通常利用する公開関数

関数	戻り値	役割
<code>LoadConfigSetting(const char *filename, LoRaConfigItem_t &config)</code>	<code>int</code>	SPIFFS上の設定ファイルを読み込み <code>config</code> へ反映する
<code>SetDefaultConfigValue(LoRaConfigItem_t &config)</code>	<code>void</code>	LoRa設定をデフォルト値で初期化する
<code>InitLoRaModule(LoRaConfigItem_t &config)</code>	<code>int</code>	設定値に従ってE220レジスタを初期化する
<code>ReceiveFrame(RecvFrameE220900T22SJP_t *recv_frame)</code>	<code>int</code>	LoRaデータを受信して受信構造体へ格納する
<code>SendFrame(LoRaConfigItem_t &config, uint8_t *send_data, int size)</code>	<code>int</code>	指定データをLoRa送信する
<code>SwitchToNormalMode()</code>	<code>void</code>	M0=0、M1=0 のNormal Modeへ移行する
<code>SwitchToWORSendingMode()</code>	<code>void</code>	WOR送信モードへ移行する
<code>SwitchToWORReceivingMode()</code>	<code>void</code>	WOR受信モードへ移行する
<code>SwitchToConfigurationMode()</code>	<code>void</code>	M0=1、M1=1 の Configuration Modeへ移行する
<code>ReadFirmwareVersion()</code>	<code>uint8_t</code>	E220ファームウェアバージョンを取得する
<code>CheckFirmwareVersion()</code>	<code>int</code>	想定したデバイス／ファームウェアか確認する
<code>init_REG09_0A_Byte()</code>	<code>int</code>	Strict Mode用REG09/REG0A管理値を初期化する

17.3 設定ファイル処理用の内部関数

以下は CLoRa の `private` 関数です。

関数	戻り値	役割
<code>OpenConfigFile(const char *filename, LoRaConfigItem_t &config)</code>	<code>int</code>	設定ファイルを開き、各項目の読み込み処理を行う
<code>ReadConfigValue(const char *key, const char *val)</code>	<code>int</code>	設定ファイルのキーと値を解析する
<code>SetConfigValue(LoRaConfigItem_t &config)</code>	<code>int</code>	解析済み設定値を <code>LoRaConfigItem_t</code> へ変換・設定する

関数	戻り値	役割
<code>SetDefaultConfigValue_into_private_val()</code>	<code>void</code>	ライブラリ内部の設定作業用変数をデフォルト値へ戻す

17.4 Strict Mode関数

以下の関数は、

```
FIRMWARE_VERSION == 0x20
```

の場合に使用可能な Strict Mode 関連関数です。

本基板は、E220-900T22S (JP) R2を搭載しているため、`firmware.h` は `FIRMWARE_VERSION 0x20` となります。

関数	戻り値	役割
<code>Write_RegisterLIB(const char *str, int str_length, uint8_t SetData)</code>	<code>int</code>	REG09/REG0Aなどのレジスタへデータを書き込む
<code>Read_RegisterLIB(uint8_t address, uint8_t bytes)</code>	<code>int</code>	Configuration Modeでレジスタを読み込む
<code>Read_RegisterLIB_s(uint8_t address, uint8_t bytes)</code>	<code>int</code>	Strict Mode中に読出し可能なレジスタを読み込む
<code>read_id_num(char RegNum)</code>	<code>uint8_t</code>	個体番号レジスタを読み込む
<code>write_id_num(char RegNum, uint8_t IDdata)</code>	<code>uint8_t</code>	個体番号レジスタへ書き込む
<code>SetConfigStrictMode()</code>	<code>int</code>	Strict Mode用REG09/REG0Aを設定する
<code>Lowvoltage_operation_Enable()</code>	<code>int</code>	低電圧動作設定を有効にする
<code>Lowvoltage_operation_Disable()</code>	<code>int</code>	低電圧動作設定を無効にする
<code>StrictModeStart()</code>	<code>int</code>	REG09 Strict Modeビットをセットする
<code>StrictModeEnd()</code>	<code>int</code>	REG09 Strict Modeビットを解除する
<code>ReceivePKTaddADRS()</code>	<code>int</code>	受信パケットへのアドレス情報付加を有効にする
<code>ReceivePKTdelADRS()</code>	<code>int</code>	受信パケットへのアドレス情報付加を無効にする

関数	戻り値	役割
<code>SendPKTaddCS()</code>	<code>int</code>	送信パケットへのチェックサム付加を有効にする
<code>SendPKTdelCS()</code>	<code>int</code>	送信パケットへのチェックサム付加を無効にする
<code>ReceivePKTaddCS()</code>	<code>int</code>	受信パケットのチェックサム処理を有効にする
<code>ReceivePKTdelCS()</code>	<code>int</code>	受信パケットのチェックサム処理を無効にする
<code>ReceivePKTaddSIZE()</code>	<code>int</code>	受信パケットのサイズ情報付加を有効にする
<code>ReceivePKTdelSIZE()</code>	<code>int</code>	受信パケットのサイズ情報付加を無効にする
<code>REG09_ByteWrite()</code>	<code>int</code>	保持している <code>REG09_Byte</code> を REG09へ書き込む
<code>REG09_ByteWrite(uint8_t setdata)</code>	<code>int</code>	引数の値をREG09へ書き込む
<code>REG0A_ByteWrite()</code>	<code>int</code>	保持している <code>REG0A_Byte</code> を REG0Aへ書き込む
<code>REG09_Set_AUX_HoldTime(uint8_t bt)</code>	<code>int</code>	AUX Low保持時間を設定する
<code>REG0A_Set_CarrierSenseTime(uint8_t ms)</code>	<code>int</code>	Carrier Sense時間を設定する
<code>Check_BufferedReceiveData_Exist()</code>	<code>int</code>	レジスタアクセス中に受信した LoRaデータの有無を確認する

17.5 Strict Mode内部処理

以下はライブラリ内部で使用する `private` 関数です。

関数	戻り値	役割
<code>Skip_Receive_LoRa_Response()</code>	<code>int</code>	レジスタ操作時の応答／受信データ処理を整理する
<code>CheckAndGetPacket(...)</code>	<code>int</code>	Strict Modeの受信バッファからパケットを検査・抽出する

18. ライセンス

本プロジェクトのソースコードおよびドキュメントは、MIT License のもとで公開しています。

個人利用・商用利用を問わず、MIT License の条件に従って、自由に使用・複製・変更・再配布することができます。

詳細については [LICENSE](#) ファイルを参照してください。

19. サポートについて

本プロジェクトは無償公開のサンプルおよびライブラリであり、技術サポートは提供していません。

本プロジェクトのソースコードおよびドキュメントは現状のまま（AS IS）で提供されます。
使用環境や接続機器などによる動作を保証するものではありません。

本プロジェクトの利用、変更、組み込み等は、利用者自身の責任において行ってください。

なお、本プロジェクトのソースコード、README およびその他のドキュメントは、予告なく変更される場合があります。

ソースコードの変更にあたっては、可能な限り既存バージョンとの互換性を維持する方針としています。

改訂履歴

日付	内容
----	----

2026-09-01	初版
------------	----
