BC7215AC Arduino

Universal

Air Conditioner Remote Control Library

V4.x

Index

Introduction

The BC7215AC Arduino Universal Air Conditioner Remote Control Library (hereinafter referred to as the AC Remote Control Library) is a dedicated code library designed for Arduino systems. It is derived from the C-language implementation of the BC7215A offline air conditioner remote control code library and has been adapted to accommodate the specific characteristics of Arduino platforms. This library requires the BC7215A chip for operation; consequently, it incorporates both the BC7215 chip driver library and the universal air conditioner remote control functionality, requiring only a single installation by the user. The V4.x version encompasses the majority of air conditioner brands and models, with continuous iterative enhancements.

In contrast to libraries reliant on sample databases, this library employs an encoding rule-based methodology. Configuration is achieved not through brand or model selection but via automatic protocol identification from decoded infrared signals, facilitating database-independent universal control. Although thousands of air conditioner brands and models exist, the underlying remote control protocols are limited to approximately one hundred variants. Protocol sharing across brands, including major manufacturers, is prevalent. This rule-based universal remote control approach provides the following advantages:

- Independent of networks and databases; fully offline operation suitable for resource-constrained microcontrollers.
- Inherent compatibility with future models, as air conditioner infrared remote functions are largely standardized, and new products typically adopt established protocols.
- Enhanced support for less common brands, which frequently utilize mature OEM solutions from established vendors.

This library operates without internet connectivity and depends solely on the BC7215A chip, with no additional external dependencies. It is compatible with any Arduino system possessing adequate program memory and RAM. Accompanying examples demonstrate usage on ESP8266 and ESP32 processors.

Leveraging the BC7215A's universal decoding capabilities, the library implements a protocol-rule-driven general-purpose driver, as opposed to a waveform database. This design ensures compactness and offline functionality. The library supports four fundamental air conditioner control operations:

- Temperature adjustment
- Operating mode selection
- Fan speed control
- Power on/off management

Air Conditioner Remote Control Protocol Overview

Infrared remote controls for air conditioners are categorized into two primary types. Fixed-code remotes transmit invariant infrared codes per button and are identifiable by the absence of a LCD display on the remote. These resemble standard audiovisual equipment remotes and can be replicated using the BC7215(A)'s core functions without a specialized library; refer to the "learning remote control" example in the BC7215 Arduino driver library.

The predominant type involves variable-code remotes, featuring extended signal lengths that encapsulate comprehensive control data, including set temperature, operating mode, and fan speed. These are distinguished by the presence of a LCD display. Codes from identical buttons vary with the device's state, and encoding differs by manufacturer and model. Signal generation for such remotes necessitates this AC Remote Control Library.

BC7215AC Arduino Air Conditioner Remote Control Library Usage Instructions

Note: Given the library's dependence on the BC7215A chip, users are advised to familiarize themselves with the chip's operational principles, data formats, and driver functions prior to proceeding. This knowledge facilitates comprehension. Relevant documentation is available in the library's documentation directory.

The library's utilization is straightforward, comprising three essential steps for controlling any air conditioner.

Step 1: Sampling the Original Air Conditioner Remote Control Signal

Employ the BC7215A's decoding feature to capture a designated signal from the target air conditioner's remote: cooling mode at 25°C (77°F). The sampled data enables the library to analyze and identify the original signal format.

Note: Refrain from using universal remotes as signal sources, as they often emit multiple protocols sequentially per button press. If the effective protocol is not the initial one, erroneous data may be captured.

Step 2: Initializing the Remote Control Library

Utilize the decoded data as input parameters for the library's initialization function to recognize the air conditioner's encoding format.

Step 3: Configuring Temperature, Mode, and Fan Speed

Invoke the library's setting functions to transmit infrared signals and establish the air conditioner in the desired state.

Additional Functions: Power On/Off

The library includes dedicated functions for powering the air conditioner on or off.

Recommended Practices

Following successful initialization, store the initialization data for subsequent use. Upon system startup, load the stored data to perform initialization, obviating the need for repeated sampling and enabling immediate operability. Consult the example programs for implementation details.

System Requirements

The library is applicable to any Arduino system with sufficient program storage and RAM capacity. Demonstration programs utilize ESP8266 and ESP32 platforms.

Detailed Description of the Air Conditioner Remote Control Library

Library Composition

The BC7215AC library comprises two distinct driver components: the BC7215(A) chip driver library for fundamental chip control, and the specialized air conditioner remote control library. The chip driver library includes four demonstration programs:

- Arbitrary remote control decoding
- Learning-type remote control
- Dual infrared remote switch
- Infrared data transmission

Air conditioner applications mandate the BC7215A variant, as the standard BC7215 is unsuitable for air conditioner signal decoding. The air conditioner remote control library provides eight demonstration programs:

- ESP8266 blocking version (software serial communication with BC7215A; Arduino IDE serial monitor for interaction; blocking design with straightforward logic)
- ESP8266 non-blocking version (software serial communication with BC7215A; Arduino IDE serial monitor for interaction; non-blocking design)
- ESP32 serial monitor English version (port of ESP8266 demonstration)
- ESP32 serial monitor Chinese version (port of ESP8266 demonstration)
- ESP32 LCD English version (utilizes LILYGO T-Display onboard buttons and LCD)
- ESP32 LCD Chinese version (utilizes LILYGO T-Display onboard buttons and LCD)
- ESP32 MQTT English version (extends LCD version with networking for MQTT-based control and status reporting)
- ESP32 MQTT Chinese version (extends LCD version with networking for MQTT-based control and status reporting)

Hardware Connections

The BC7215A chip serves as the core component for air conditioner control, interfacing with the Arduino via serial communication. Two auxiliary signals are required: MOD (Arduino output for transmit/receive mode selection) and BUSY (BC7215A output for serial data transfer control).

Interface Functions

The following delineates the interface functions of the BC7215AC air conditioner remote control library. For details on the chip driver library functions, consult the BC7215 Arduino driver library manual.

1. Start/Stop Sampling Functions

```
startCapture();
```

```
stopCapture();
```

These functions initiate and terminate the BC7215A chip's receive mode for sampling air conditioner remote signals prior to library initialization.

2. Sampling Status Query Functions

```
bool signalCaptured();

bool signalCaptured(bc7215DataVarPkt_t* targetData,  bc7215FormatPkt_t* targetFormat);
```

These functions ascertain whether a complete signal has been sampled. The parameterless variant stores results in the library's internal buffer; the alternative stores in user-designated locations. Returns true upon completion.

3. Initialization Functions

```
bool init();

bool init(const bc7215DataMaxPkt_t& data, const bc7215FormatPkt_t& format);

bool init(const bc7215DataVarPkt_t* data, const bc7215FormatPkt_t* format);

bool init(uint8_t cnt, bc7215DataMaxPkt_t const data[], bc7215FormatPkt_t const
format[]);
```

These functions initialize the library using sampled data. The simplest form employs the internal buffer, suitable for being called right after a capture has been done. The 2nd and 3rd accepts explicit data and format inputs, suitable for restored initialization data. The third addresses specialized multi-segment protocols (refer to extraSample()).

Returns true on successful initialization. Input data must originate from cooling mode at 25°C; utilize the fan speed button to maintain state consistency.

4. Air Conditioner Setting Functions

```
const bc7215DataVarPkt_t* setTo(int tempC, int mode = -1, int fan = -1, int key = 0);
```

Post-initialization, this function configures the air conditioner state. Parameters include temperature, mode, fan speed, and key (for protocols incorporating button-specific encoding).

Default values permit sequential omission. Out-of-range values imply no change for the respective parameter.

Invocation triggers immediate infrared transmission; returns a pointer to the transmitted data.

Temperature (temp):

Range: 16-30 (corresponding to 16°C-30°C; For Fahrenheit, users must convert to Celsius themselves). Exceeding range: no change.

Mode (mode):

Range: 0-4 . Exceeding range: no change.

0 - Auto

1 - Cool

2 - Heat

3 - Dry

4 - Fan

Fan Speed (fan):

Range: 0-3 . Exceeding range: no change.

0 - Auto

1 - Low

2 - Med

3 - High

Key (key):

Range: 0-3. Exceeding range: no change.

0 - Temperature +

1 - Temperature -

2 - Mode

3 - Fan

Note: The library generates protocol-compliant packets without validating setting validity. Air conditioners impose model-specific constraints (e.g., temperature irrelevance in fan mode, or some models are cooling only without heating mode). Adhere to target device limitations; invalid settings may yield unpredictable behavior.

5. Air Conditioner Power On/Off Functions

```
const bc7215DataVarPkt_t* on(void);

const bc7215DataVarPkt_t* off(void);
```

These functions manage power state. Invocation transmits the corresponding signal; returns pointer to data to be sent.

Most models activate upon state-setting commands in off state, reverting to prior or initialization state. Select models require dedicated ON command, followed by setTo() for state configuration.

6. Find Next Matching Function

```
bool matchNext();
```

Initialization automatically matches protocols; however, similar variants may exist, causing discrepancies. This function searches for alternative matches. Returns true on success; test functionality. Repeat until false.

Callable only post-successful initialization; revert via re-initialization.

7. Predefined Protocols

Certain protocols elude direct BC7215A decoding, necessitating waveform capture and online conversion. The library includes pre-converted data. Employ if standard sampling/initialization fails.

Associated functions:

Acquire Number of Predefined Protocols

```
uint8_t cntPredef();
```

Returns the count of embedded predefined protocols.

Acquire Name of a Predefined Protocol

```
const char* getPredefName(uint8_t index);
```

Returns mnemonic name for specified index.

Initialize Using a Predefined Protocol

```
bool initPredef(uint8_t index);
```

Initializes with embedded protocol by index.

8. Acquire Library Version Information

```
const char* getLibVer();
```

Returns a string containing the library version.

9. Determine if Additional Sampling is Required

After successful initialization, call

```
uint8_t extraSample();
```

This function returns an 8-bit integer indicating if the current initialized protocol has commands using special formats different from ordinary commands. If so, additional sampling is needed for those commands. The return value may be 0, 1, 2, 3, or 4:

0 - No special format needed

1 - Temperature setting command needs special format

2 - Mode setting command needs special format

3 - Fan speed setting command needs special format

4 - The protocol uses multi-segment infrared signals and requires initialization with multiple sampled data

10. Store Additional Format Information

If the above function returns a non-zero result, the user needs to perform additional sampling for the command requiring the special format to allow the A/C library to understand the special format information. After sampling for that special command, use

```
bool saveExtra(const bc7215DataVarPkt_t* data, const bc7215FormatPkt_t* format);
```

to store it in the air conditioner remote control library. The input data is obtained during sampling via

```
bool signalCaptured(bc7215DataVarPkt_t* targetData,  bc7215FormatPkt_t* targetFormat);
```

Returns true if saved successfully, false otherwise.

If extraSample() returns 4, the protocol uses a multi-segment infrared signal format and requires reinitialization with multiple sampled data; see section 14 for details.

11. Retrieve Base Data Packet and Base Format Packet

```
const bc7215DataVarPkt_t* getDataPkt();

const bc7215FormatPkt_t*  getFormatPkt();
```

Obtain current initialization data for storage and future use.

12. Retrieve Additional Format Information

Analogous to base retrieval; for protocols with extras. Return structure:

```
typedef struct {
    uint16_t    bitLen;     /**< Bit length placeholder, always set to 0 */
    union {
        uint8_t data[1];    /**< Raw data array */
        struct {
            const bc7215FormatPkt_t*    fmt;        /**< Pointer to format packet */
            const bc7215DataVarPkt_t*   datPkt;     /**< Pointer to data packet */
        } msg;       /**< Message structure containing format and data packets */
    } body;          /**< Union containing either raw data or structured message */
} bc7215CombinedMsg_t;
```

Refer to examples for usage.

13. Query BC7215A Chip Status

```
bool isBusy();
```

Returns true during signal transmission/reception. Post-transmission, await completion and >100ms interval before subsequent signals to ensure proper reception.

14. Multi-Segment Data Initialization

If extraSample() returns 4, the remote format differs from usual, being multi-segment, requiring resampling to capture multi-segment signals. During sampling, use

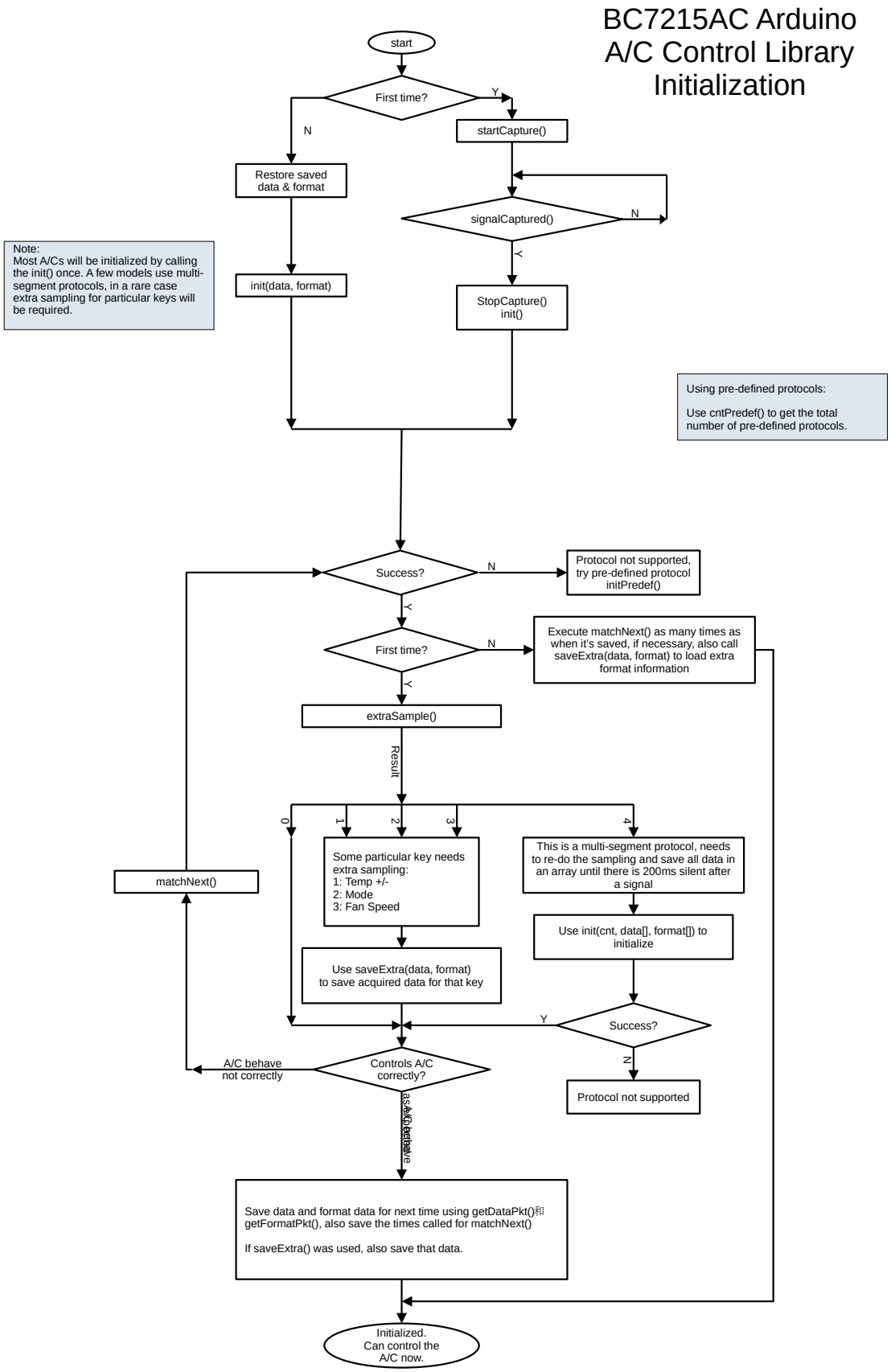
```
bool signalCaptured(bc7215DataVarPkt_t* targetData, bc7215FormatPkt_t* targetFormat);
```

for querying, storing each sampled data and format in arrays, then initialize with

```
bool init(uint8_t cnt, bc7215DataMaxPkt_t const data[], bc7215FormatPkt_t const
format[]);
```

After each signal is sampled, start a timer and query isBusy(). If a new signal is received within 200ms, save it and restart the timer. Stop sampling only when isBusy() remains false for >200ms continuously, then initialize the air conditioner remote control library with all sampled data, where cnt is the number of signals received.

Programming Flowcharts



A/C Control Flowchart