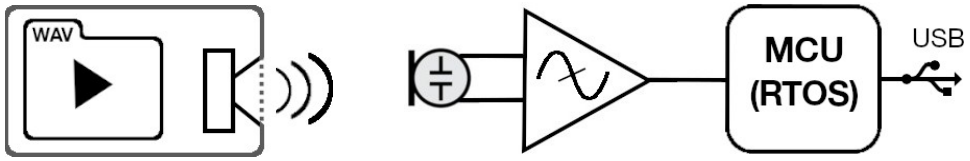
Audio Code Receiver Application

A microcontroller application running under FreeRTOS that extracts a text message from a digital input and writes it to a USB port.

The original text message is encoded into an audio .wav file, which is played through a speaker.

The resulting audio is detected by the audio code receiver, amplified and then filtered to produce a digital input to a microcontroller.



Application Goal

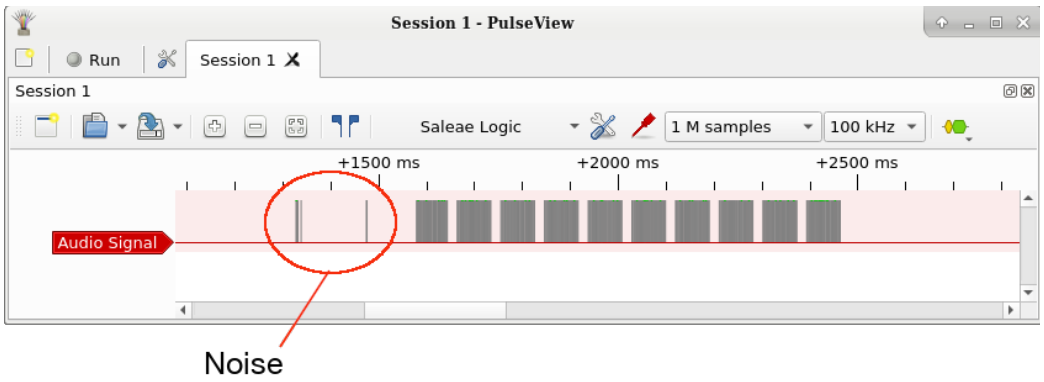
Transform a noisy input signal into an accurate copy of the original text message, using the microcontroller resources efficiently and effectively.

To meet the application goal, FreeRTOS was chosen, running on an ESP8266.
(The ESP8266 already runs FreeRTOS as part of it's Wi-Fi system).

Digital Input

The digital input consists of the same message repeated multiple times. Each message consists of two components followed by an inter message gap.

The first component is used to alert the receiver that a message is being sent.
The second component is the message content, which consists of frames of 10 bits.

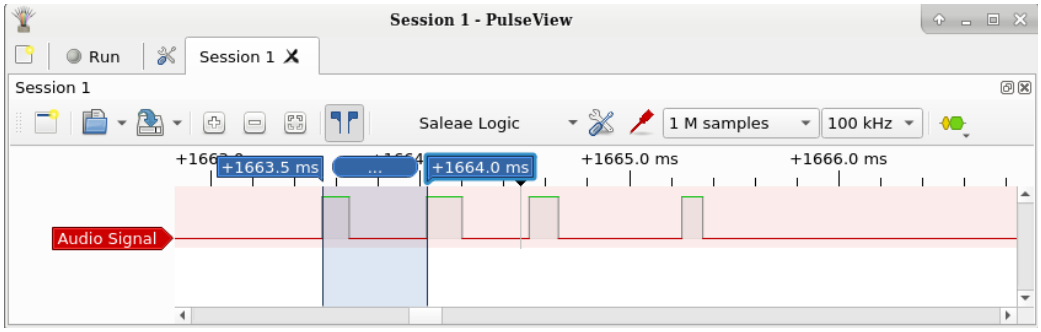


The logic analyser snapshot above shows the digital input (Audio Signal). The message is repeated 10 times with inter-message gaps. Notice the noise that precedes the messages. This has to be filtered out by the software.

The inter-message gaps are used to determine when each message is complete.

Alert Component

In order to detect the start of a genuine message, a series of precisely timed pulses precede the message content.



The logic analyser snapshot above shows three pulses spaced 0.5 ms apart. The fourth pulse is the start of the first frame of the message contents.

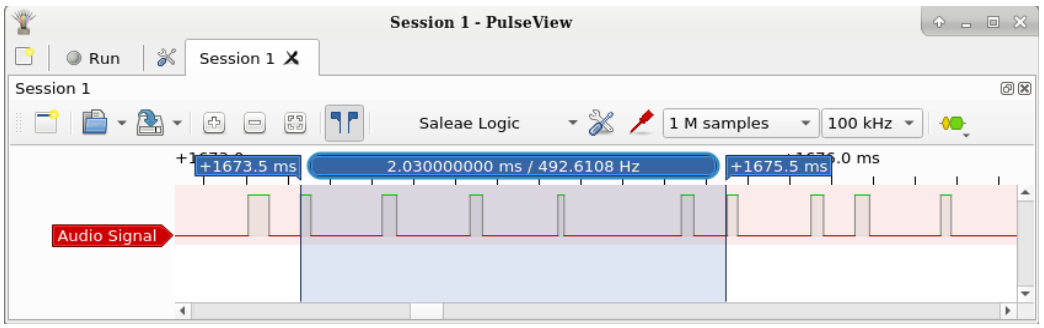
The Message Detector function of the Message Build Task (see below), waits for the three pulses and if detected, flags the presence of a message. The timing of the alert pulses is not repeated within the message content.

Contents Component

The contents consists of a series of frames. The first few frames contain the same character and are designed to synchronise the receiver's electronics to the timing of the data input pulses. The following frames contain the length and ASCII content. The last frames contain the message's checksum.

Frame Format

Each frame contains 10 bits. The first bit is the start bit and is always a '1'.
The next 8 bits are the actual data. The last bit is an even parity bit.



The logic analyser snapshot above shows a 10 bit frame.

Each bit in the source .wav file occupies 200 µs. Looking at the analyser's output, the frame has taken 2.03 ms (2030 µs), instead of 2.00 ms. This is a result of delays introduced by the speaker/microphone/receiver electronics. Also of note is the difference in duration of the '1' bits.

Software Techniques for the Removal of Noise

Noise Preceding Message

The message detector ignores data input until the alert signature is recognised.

Noise within Message Content

The frame bits are spaced 200µs apart. By sampling the data input at approximately 200 µs intervals, any noise present in between is ignored.

Noise due to Varying Duration of ‘1’ Bits

The rising edge of each bit is used to set a start of bit timestamp. By detecting the rising edge only, the actual duration of the bit is ignored.

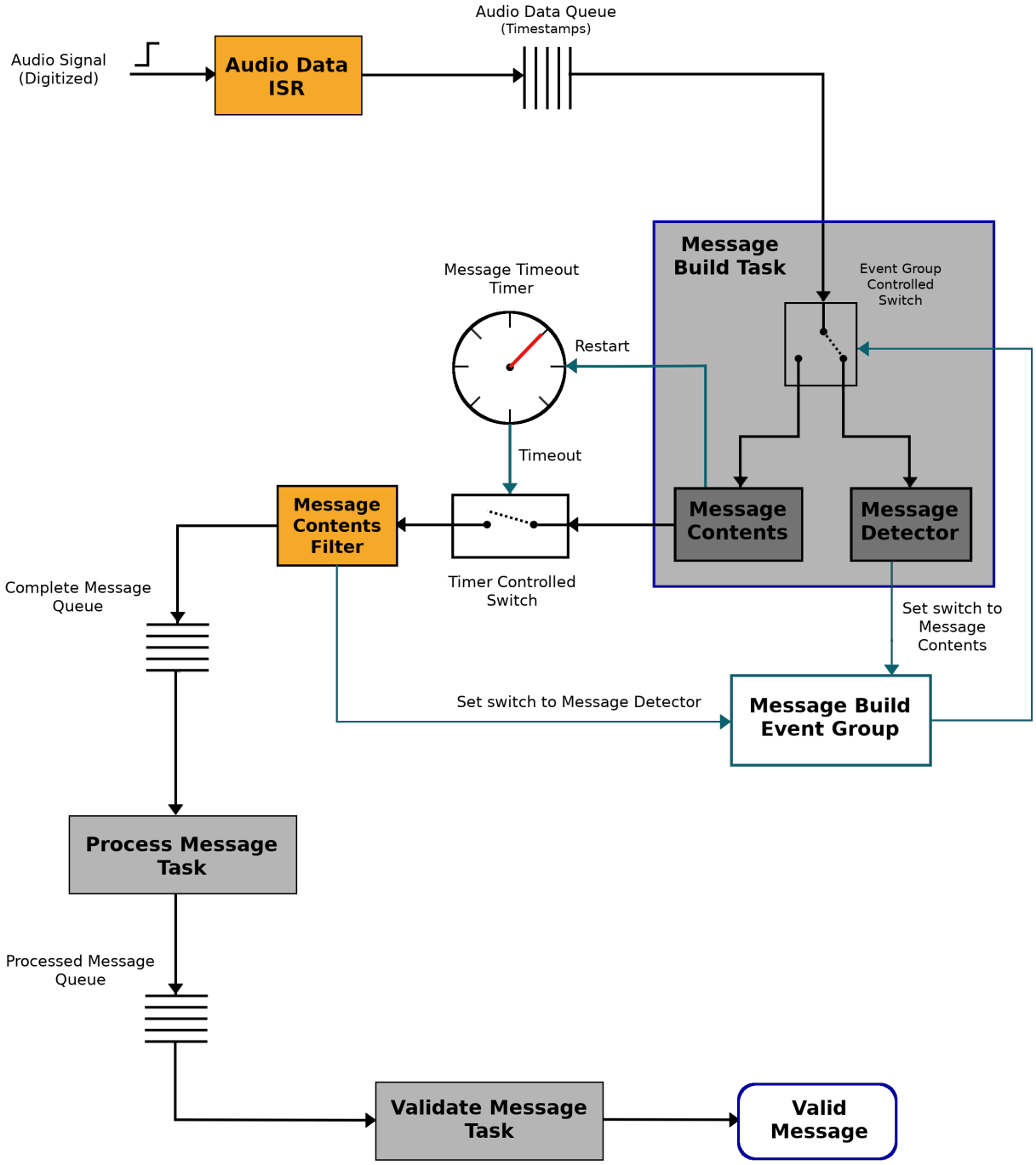
Corrupt Data

A combination of parity bits and message checksum are used to indicate corrupt message content.
By repeating the same message multiple times, different copies can be compared, possibly resulting in a valid copy.

RTOS Application

The application was written using C and C++ using FreeRTOS and the Expressif IDF.

The diagram below shows the data flow and the RTOS elements of the application:



The two switches (Timer controlled switch and Event Group controlled switch) represent points where the data flow is controlled. They are not RTOS elements.

The application can be separated into three stages:

1. Capturing data
2. Building a message
3. Validating the message

Capturing Data

The audio signal is connected to a general purpose input on the ESP8266 microcontroller. The input is configured to generate an interrupt when it receives a rising edge and is attached to an Interrupt Service Routine (ISR). The ISR places the timestamp of the rising edge on the Audio Data Queue.
The timestamps are then interpreted to build the message.

The ISR is very short and is executed immediately, taking precedence over the RTOS tasks. The queue acts as buffer, ensuring that no rising edge is missed.

Building a Message

Each message is composed of an alert component and a contents component.
The Message Build Task reads the timestamps from the Audio Data Queue and then routes them to either the Message Detector (alert) or the Message Contents processes.

Initially, the Message Build Task is attempting to detect a message, so the timestamps will be analysed by the Message Detector.

When a message is detected (the alert signature is recognised), the timestamps are read by the Message Contents process.

An event group controlled switch routes the data flow to the Message Detector or the Message Contents process. The event group has two flags (known as bits):

1. Message Detector bit - route timestamps to Message Detector
2. Message Contents bit - route timestamps to Message Content

Setting one bit, resets the other bit.

The Message Timeout Timer is used to detect the end of message gap (multiple frames in duration). It is re-started for each frame ‘1’ bit received.

When it times out, the message contents built by the Message Build Task are sent to the Message Contents Filter. If the message contents are suitable for further processing, they are placed on the Complete Message Queue.

The event group’s Message Detector bit is then set, allowing the Message Build Task to look for the next message.

Validating the Message

The Process Message Task reads the complete message from the Complete Message Queue and performs the following checks:

- Message length match
- Message checksum match
- Message byte parity match

Any mismatches together with the complete message are placed onto the Processed Message Queue.

The Validate Message Task reads the entries from the Processed Message Queue and if a message with no mismatches is found, marks it as a valid message and stops the application.

If only messages with mismatches are found, it attempts to compose a valid message from all of the message copies.

RTOS Elements

The following RTOS elements are used:

- Interrupt Service Routine (ISR)
- Queue
- Task
- Timer
- Event Group

The use of the above RTOS functionality allowed the application to behave as an information pipeline with successive stages (tasks) being separated by buffers (queues).

Queues

Each application queue acts a data buffer between tasks. The queues are lightweight, only containing a list of pointers to values, structures or objects. The task reading its queue only executes when there are items left on the queue.

When the queue is empty, the task waits (blocks). This allows other, lower priority tasks to execute.

Task Priorities

Each application task is assigned a priority that is used by the RTOS scheduler to decide which task executes.The priorities are used to ensure that the highest priority task that is not blocked (waiting on an empty queue) is always being executed.

This mechanism ensures that messages are built before they are validated and that the validation takes place when the audio signal is not active.

A task with a higher priority is executed before a task with a lower priority.

Order of task execution by priority:

Task	Priority
Message Contents Filter	3
Message Build	2
Process Message	1
Validate Message	1

Timer

The timer acts as a watchdog, resetting the Message Build task, when there is a gap in audio signal activity.
This ensures that the application will be waiting for the next message.

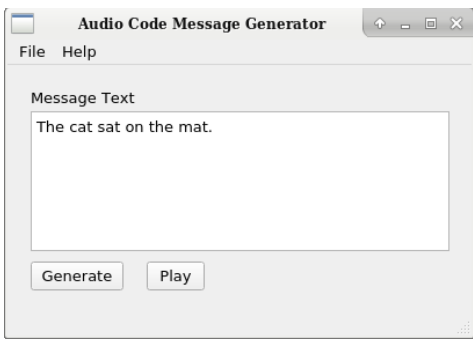
Event Group

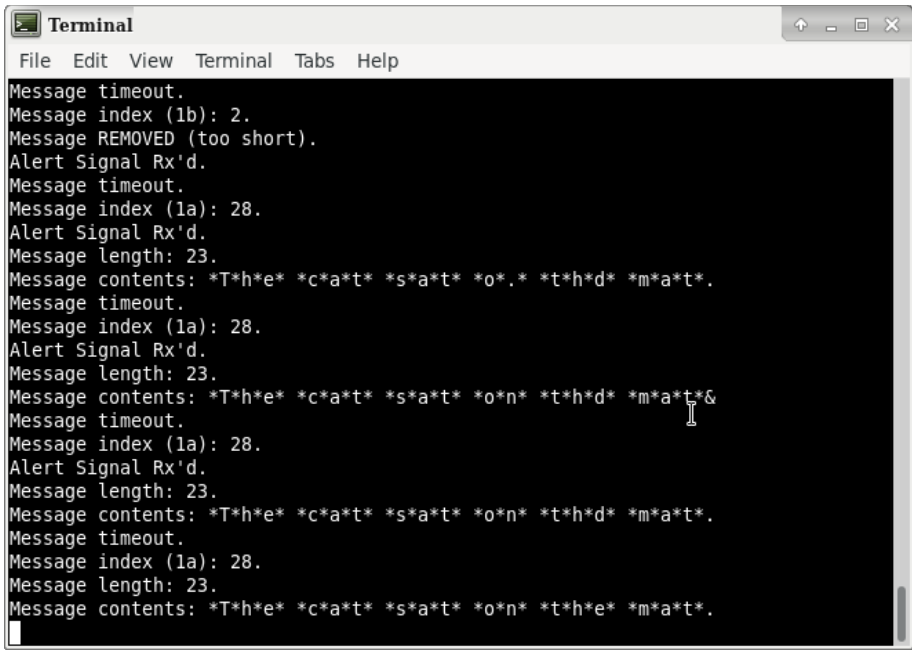
The event group consists of two flags, which can be set outside of the Message Build task. It is used to control the data flow to either the Message Detector or the Message Contents processes.
Using the RTOS event group element allows the event group flags to be set atomically i.e. without interruption. This ensures the integrity of the flags.

Results

The test system used a Linux desktop computer running a message generator/player connected to a speaker.
The Expressif IDF monitor ran on the same machine and was connected to the ESP8266 board via USB.

(The message generator front end was written using Qt/C++. The back end that produced the .wav files was written in C++).





The upper window above shows the message generator with the message “The cat sat on the mat”.

When the Play button is pressed, the message is played 10 times.

The lower window above shows the debug output of the ESP8266 running the audio receiver application.

There are five messages displayed, only the fifth message (at the bottom) has been correctly received. (An asterisk was added before each message character).

The first message was removed by the Message Contents Filter because it was too short.

Conclusions

Overall, the application was successful in extracting text messages from a noisy input signal. Using FreeRTOS with the ESP8266 enabled a robust, reliable application to be written.