Implementation Report: Module 3 - Audio Support (Text-to-Speech)

Project: Inclusive Reading Aid for Dyslexic & Visually Impaired Users **Module:** Point 3 - Audio Support (Text-to-Speech) **Status:** Completed **Date:** 2025-11-08

1.0 Introduction

This report details the technical methodology for constructing the **Audio Support** module. This module is the primary audio backend of the application, responsible for converting on-screen text into natural-sounding speech.

The objective is to provide a functional Text-to-Speech (TTS) engine that powers the "Read with Highlights" and "Listen Only" modes (defined in Module 6). This implementation involves not only generating speech but also solving the critical challenge of synchronizing speech playback with real-time text highlighting in a thread-safe manner, ensuring a non-blocking user interface.

2.0 Core Technologies & Dependencies

This module introduces several key backend libraries:

1. **pyttsx3:** The core Python library for speech synthesis. It is used to generate speech from text and provides hooks to monitor playback events (e.g., which word is being spoken).
2. **threading:** A standard Python library. It is **critically required** to run the TTS engine's blocking `runAndWait()` loop in a separate worker thread. This prevents the entire GUI from freezing during speech.
3. **PyQt6.QtCore:**
   - `pyqtSignal` : The "bridge" for thread-safe communication. A custom signal is defined to send data from the `pyttsx3` worker thread to the main GUI thread.
4. **PyQt6.QtGui:**
   - `QTextCursor` , `QTextCharFormat` , `QColor` : Used by the GUI thread to programmatically select text in the `QTextEdit` widget and apply the "highlight" style (e.g., a yellow background).

3.0 Implementation Methodology

The TTS engine was integrated by initializing it, establishing a thread-safe communication channel for highlighting, and connecting it to the existing UI controls from Module 6.

3.1 Engine Initialization

1. **Instantiation:** A `pyttsx3.init()` engine is instantiated in the `__init__` constructor and stored as `self.tts_engine` . This is wrapped in a `try...except` block to handle potential OS-level errors (e.g., no speech driver found).
2. **Highlighting Formats:** `QTextCharFormat` objects are created to define the "highlighted" state (yellow background, black text) and "normal" state (default colors).

3.2 Thread-Safe Highlighting (Signal & Slot)

This is the most complex component of the module, designed to prevent UI freezes and thread-related crashes.

1. **Custom Signal:** A class-level `pyqtSignal(int, int)` named `word_highlight_signal` is defined. It is designed to carry the `location` (int) and `length` (int) of the word being spoken.
2. **TTS Event Hook:** The `pyttsx3` engine's `'started-word'` event is connected to a private handler function, `_on_word_started`.
3. **Worker Thread Function (`_on_word_started`):** This function is executed by the `pyttsx3` **worker thread**. Its sole responsibility is to emit the `self.word_highlight_signal` with the `location` and `length` data it receives. It **does not** touch the UI.
4. **Main Thread Slot (`highlight_word`):** The `word_highlight_signal` is connected to the `highlight_word` slot. This function is executed by the **main GUI thread**. It safely performs the following:
    - Clears any previous highlighting using a `QTextCursor`.
    - Creates a new `QTextCursor` at the specified `location`.
    - Moves the cursor, selecting the word (using `location + length`).
    - Applies the `self.highlight_char_format` to the selection.

3.3 Audio Control Logic

The UI controls from Module 6 are now connected to the engine.

1. **Speed Control:** A new slot, `update_tts_speed()`, is created. It is connected to the `speed_slider` 's `valueChanged` signal. It maps the slider's value (50-200) to a valid rate for the `pyttsx3` engine (e.g., 100-400 words-per-minute) and sets it using `self.tts_engine.setProperty('rate', ...)`.
2. **Playback Logic:**
    - `_play_tts_thread(self, text)` : A function designed to run in a separate thread. It contains the blocking calls: `self.tts_engine.say(text)` and `self.tts_engine.runAndWait()`.
    - `start_tts(self)` : A function that retrieves the current text from `self.text_area` and starts a new `threading.Thread` targeting `_play_tts_thread`.

3.4 Mode Integration (Finalizing Module 6)

The `update_mode()` slot (from Module 6) is now finalized to control playback:

- **"Read Only":** Calls `self.tts_engine.stop()` to immediately halt any active speech.
- **"Read with Highlights":** Calls `self.text_area.show()` and `self.start_tts()`.
- **"Listen Only":** Calls `self.text_area.hide()` and `self.start_tts()`.

Finally, a `clear_highlighting()` function is called whenever a new file is loaded (e.g., in `open_text_file()`) to reset the UI.

4.0 Summary of Features Implemented

- [x] **Text-to-Speech:** `pyttsx3` engine is fully integrated and generates speech.

- [x] **Non-Blocking UI:** `threading` is used to run all audio playback, ensuring the UI remains responsive.
- [x] **Synchronized Highlighting:** A thread-safe signal/slot system provides real-time word highlighting synchronized with the audio.
- [x] **Adjustable Speed:** The "Reading Speed" slider from Module 6 is now functional and controls the TTS playback rate.
- [x] **Full Mode Functionality:** The "Read with Highlights" and "Listen Only" modes are now fully operational.

5.0 Conclusion

Module 3 is fully implemented. It successfully provides the core audio-visual feedback required by the project. The use of threading and PyQt's signal system solves the primary technical challenge, resulting in a stable and responsive user experience.

The application's core functionality (Input, Display, Interface, and Audio) is now complete. The next logical steps are the implementation of the text-processing modules: **Module 4 (Summarization & Simplification)** and **Module 5 (Translation)**.