Standalone Braille Writing Tutor

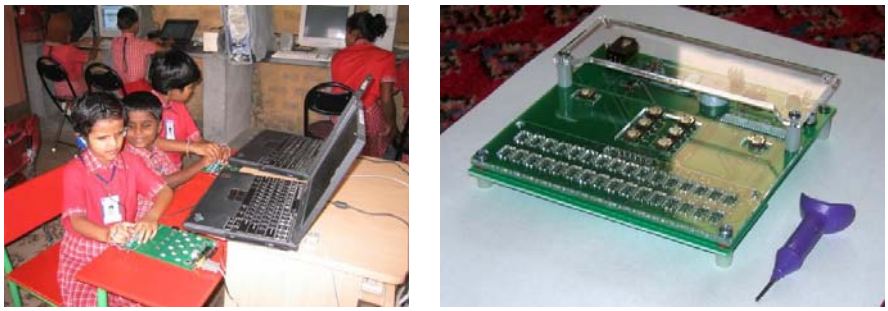
Samitha Ekanayake

January 8, 2013

Abstract

1 Introduction

Braille education has been a challenging experience for every beginner studying Braille language. In particular learning to write using the slate and stylus is highly challenging since the learner, in this case even an experienced user, cannot get an immediate feedback on what he/she has written. "The Braille Writing Tutor" (BWT) project started by Techbridge world is aimed at developing assistive technologies to fill the gap in the educational tools for braille writing and reading [**?**, **?**, **?**, **?**, **?**, **?**]. The primary objective was to develop a low-cost electronic braille interface which provides immediate audio feedback on the letter/character written. Particularly this device is intended to use in developing countries, where access to modern technology is not plentiful and the cost of the available technologies keeps them from using. Techbridge world researchers have successfully developed a low cost device which can provide audio feedback through a computer (see Figure **??**). In essence this device acts as a Braille input to the computer. The computer software then processes the user input and provides appropriate audio feedback [**?**, **?**, **?**]. The Figure **??** describes the basic functionality of the BWT.



(a) BWT Version 1 (b) BWT Verison 2

Figure 1: Previous developments of the BWT. Both versions are depending on a computer for voice feedback and for user input processing.

This device was evolved over years and the current version, BWTv2, was field tested in schools for visually impaired across the world and received valuable feedback from the intended audience [**?**, **?**, **?**]. Reliable electricity supply and finding computer to use with the braille writing tutor have been some of the major challenges encountered in the blind education schools in developing countries. During the field evaluations, every end-user/organization, although they admired the concept, suggested that this device should be able to operate on battery power and should not depend on an external computer This leads to the development of the stand-alone braille tutor. This document introduce the hardware design and and some aspects of software implementation in the new stand-alone version of the Braille Writing Tutor.

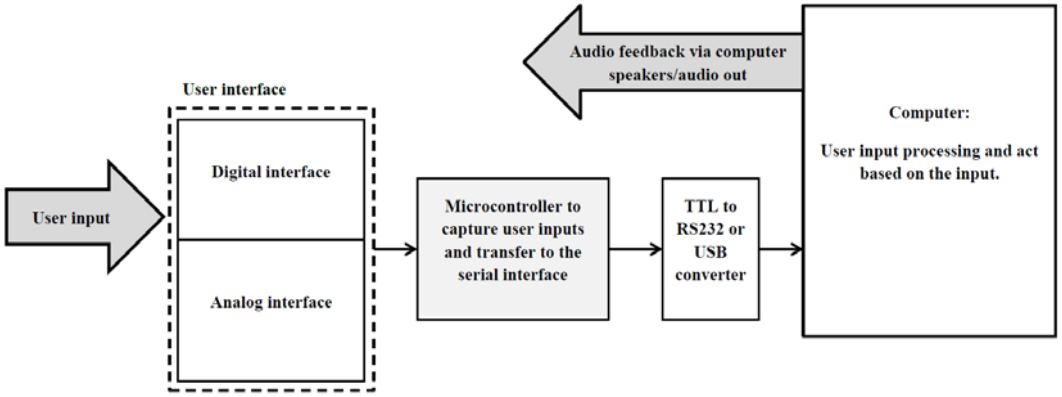


Figure 2: Overview of the Braille Writing Tutor v1 and v2 hardware

Design Requirements for stand alone version of the Braille tutor is primarily based on the user feedback from the previous versions and can be summarized as follows:
Battery requirements: Ideally the device is powered from rechargeable batteries and should be able to recharge using a common power supply. In an event where rechargeable batteries are not available, the device could be powered with non-rechargeable batteries. Also it is a requirement that the Braille tutor operates at least 6 hours (two sessions in a school) without replacing/recharging the batteries.
Mode requirements: Similar to the BWTv2, ability to accommodate multiple modes where users can select the mode from the pre-defined list of modes. Also an interface/template for developers to develop new modes and integrate with existing mode set. In addition to those, the teachers/users, in some instances need to limit the availability of the modes in the device. In particular in a class room environment teachers need to use selected set of modes for certain sessions.
Hearing requirements: Ability to operate multiple devices in the same class room is an essential requirement. This entails the devices consists of personalized audio interface which does not disturb the others.
Cost requirements: The primary goal of this project is to develop assistive technologies for developing and under-served communities which entail the cost of this device to be as low as possible while providing the required functionality.
User skill requirements: The concept of the Braille cell and the formation of various letters from the combination of braille dots is an important learning aspect for every beginner in Braille education. With conventional Braille cells, understanding the Braille cell concept and learning letters can be a tedious task, especially for small children who are starting Braille writing/reading. This entails different user interface for beginners to learn the concept of Braille cell and to practice letter formation.

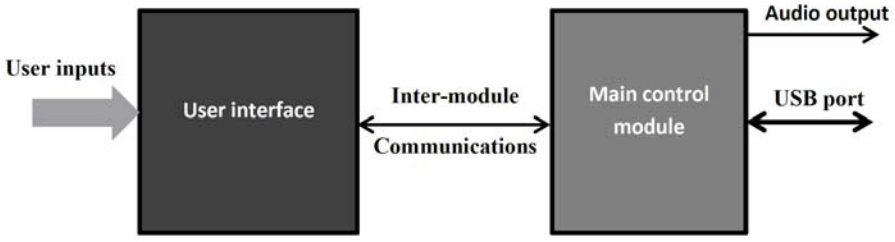


Figure 3: Two module architecture of the SABT. The user interface module essentially perform similar task as the BWTv2 device, however, we have included more on-board processing to reduce computational overhead on the main module.

To address the above requirements we came-up with a design that we call Stand Alone Braille Tutor or SABT. This design consists of two modules; main control module and the user interface. The user interface module essentially process the inputs from the user via the push buttons and the braille slate interface and inform the main control module. While the main control module processor is interfaced with many devices: user interface module, external computer, storage and audio output device. This two module architecture is developed in order to enable multiple user interfaces addressing three different user skill levels: primary, intermediate and advanced users. With this concept, the users can easily swap the user interface to suit their skill level. In particular this enables the students starting with Braille to get the required writing and reading skills from the concept of Braille cell to the sentence practice in specialized user interfaces. More details on the user interfaces are provided in the next section of this report. In 2011, a prototype of the SABT was field tested with partners in Pittsburgh, USA and in Chittagong, Bangladesh. We received positive feedback on the SABT's accessibility and its potential for educating young and elderly people with visual impairments. Based on the feedback we received from the demonstrations/field tests, we further enhance the layout of the user-interface and some of the electronics to satisfy user requirements. In this report we are explaining the technical details of the second version of SABT.

Figure 4: Getting user feedback for the version 1 of the SABT. The user comments on the version 1 was used to further enhance the second version of the device.

This report is organized as follows. In the next section we introduce basic operation of the SABT togther with layouts of the primary, intermediate and advanced user interfaces. Then we explain the electronics design of the two modules: main control unit and the user interface controller together with communication protocols we used in the design. Finally, we discuss the composition of a mode, interactions of modes with the other elements in the SABT firmware and also we introduce the mode manager utility developed for the developers and the teachers to configure the device. The appendix consists of the electronic schematic designs and the PCB layouts.

2 User interfaces and operation

SABT introduces a multiple user interface design which enables users to switch between the interfaces based on the skill level. Three interfaces were developed targeting primary, intermediate and advanced

user skills on Braille writing and reading. All three user interfaces have a common area for the control buttons. Figure **??** illustrates the layout of the common area for the control buttons and the area designated for the specialized user interface.

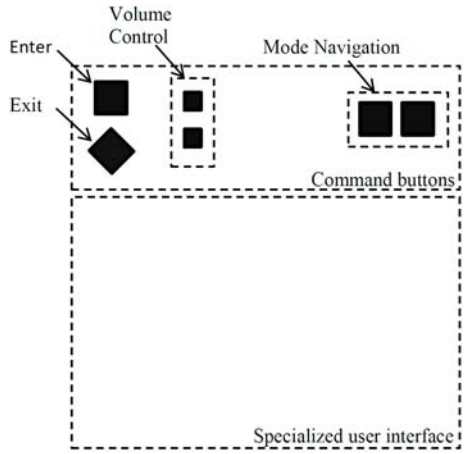


Figure 5: Overview of the user interface. The command button section is common to all three interfaces.

2.1 User interactions with control buttons

The control area/command button area consists of three button pairs: "Enter/Exit", volume "up/down" control and mode "forward/backward" navigation. Every control button can be configured to serve different purposes in the modes, however, their usage in the mode selection process is common to all interfaces. In the mode selection/navigation process, the "Enter/Exit" pair is used to go into a mode and get out of a mode. On the other hand, when the user is in a mode which requires "YES/NO" answers for questions, these buttons can be used to provide the user answers. As the name implies, the "volume control" buttons are primarily used to adjust the headphone output. As a secondary function, the "volume control" buttons are used to switch between Reading and Writing modes for the Braille cell. The "mode navigation" buttons are used to navigate the main menu. Figure **??** illustrates the user interactions with SABT via control buttons.

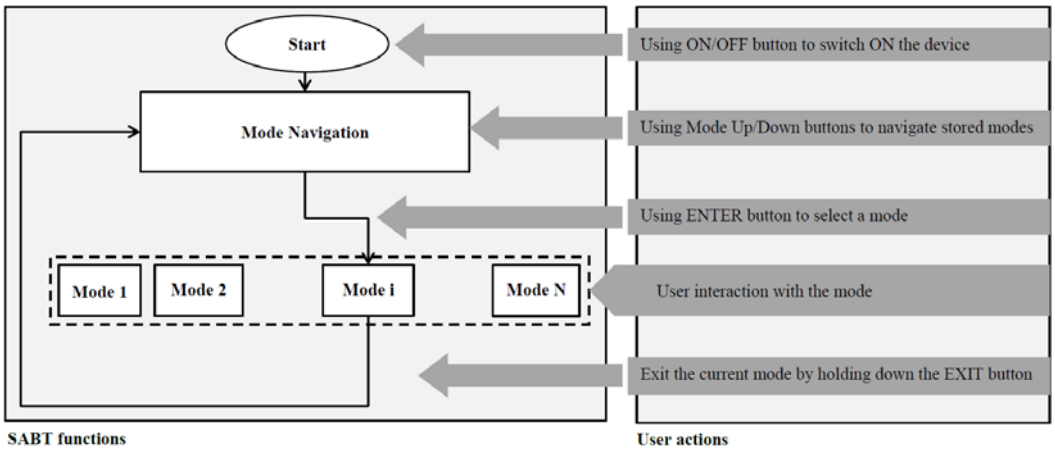
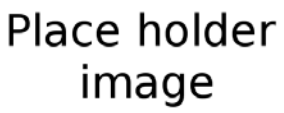


Figure 6: Using control buttons in the mode selection process

The placement and the shapes of the control buttons were based on the field test feedback of the first prototype SABT in Pittsburgh, USA and Bangladesh. The frequently accessed buttons are the Enter/Exit pair and the mode navigation pair in that order, and thus places in the left most and right most edges of the device. Also the Enter and Exit buttons were given two different shapes so that a visually impaired user can easily identify them. The volume control buttons, which are not used frequently as the others are given small button faces and located in the middle of the control area. Figure **??** compares the enhancements done for the version 2 UI based on the user feedback on the version 1.

(a) Version 1

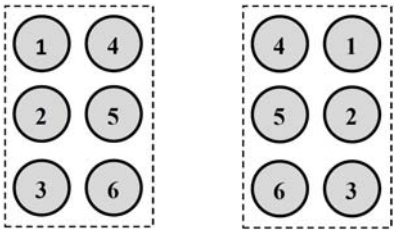


(b) Version 2

Figure 7: Comparing enhancements in the user interface layout

2.2 Braille Cell implementation

A Braille cell consists of six dots as illustrated in Figure **??**. In typical paper based Braille writing, as sub set of those six dots are embossed on the paper to represent a Braille letter/symbol. In order to read the letter/symbol the paper is flipped and read the mirror image of the dot pattern. This entail the student to learn/practice the mirror image of the alphabet for reading and writing purposes. In SABT, user can switch between Reading and Writing modes by long pressing both volume buttons.



(a) Reading mode (b) Writing mode

Figure 8: Braille cell and dot layouts for Reading and Writing modes

Different combinations of dots create letters in the Braille alphabet. For example in the English Braille alphabet; letter ‘a‘ is represented by dot 1, letter ’b’ by the combination of dots 1 and 2, letter ’c’ by dots 1 and 4 etc. In this design, pressing/activating a Braille dot generates an unique electrical signal in the circuitry and the processor in the UI module captures that signal. The electronic design is explained later in this report. A letter is generated by pressing a series of dots, and the user need to indicate the end of the dot sequence by pressing the “Enter” button. Figure **??** illustrates how the letters ’a’, ’b’ and ’c’ are generated in the SABT. Note that the dots can be pressed in any sequence to generate the same letter, for instance, 1,2,E and 2,1,E both generates the letter ’b’.

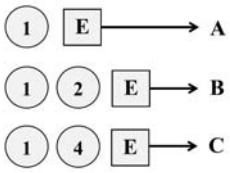


Figure 9: Writing letters: Using *enter button* to indicate the end-of-letter

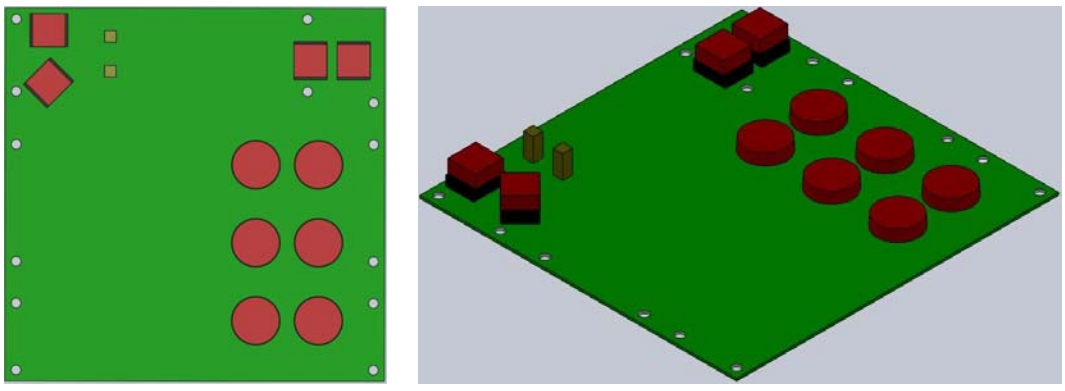
As an alternative to this approach to identify the end of the letter, one can propose several other approaches such as time-delay to detect the end-of-letter, pressing all-buttons at the same time or use the user interactions with the next cell position to identify the end of current letter. All the above approaches have several disadvantages compared to the approach implemented in SABT. For example, the time-delay approach is entirely depended on the skill level of the user, i.e. how fast he/she can do the dot sequence for a letter. Therefore this approach is not suitable for a device that is intended to use by diverse audience. The second option, which was suggested by some of the users in one of our demonstrations, is applicable only to the button based Braille cells. Moreover, this method imposes technical difficulties in the analog signal based user interface circuit. The last option can only be applied to certain user interfaces, i.e. primary interface can not use this method.

2.3 User Interface layouts

As mentioned earlier in this report, SABT is designed to accommodate multiple user interfaces. Three user-interface layouts are developed to satisfy the skill and learning requirements of primary, intermediate and advanced users. The system architecture is designed such that more layouts focused at specific user requirements (e.g. games, computer interaction etc) can be introduced to SABT.

2.3.1 Primary user interface:

The primary interface is focused on the users getting started in Braille education. It is essential that every new student for Braille education need thorough knowledge on the concept of the Braille cell. In particularly the location of the dots in the Braille cell in both reading and writing modes, and how to utilize the Braille dots to create letters. In order to satisfy that requirement, in this interface we replicate the Braille cell using large push buttons. The spacing of these buttons are proportionate to a standard slate based braille cell dot spacing.

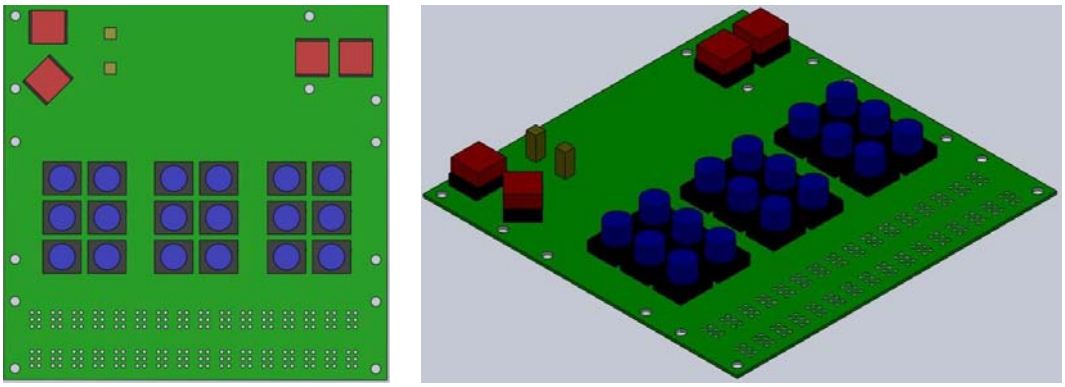


(a) Top view (b) 3D view

Figure 10: CAD screenshots of the Primary User Interface layout

2.3.2 Intermediate user interface:

Intermediate interface is developed to address the user group who have the concept of the braille cell - but still need more practice and are developing the skills in creating words and practicing letters that have more than single braille cell. This interface consists of three six-button braille cells, and two slate rows with 16 cells in each row. The button based cells are acting in the same way as the primary interface Braille cell buttons, with the exception that these button faces are small and space is reduced in order to make a smooth transition to the smaller Braille slate cells.

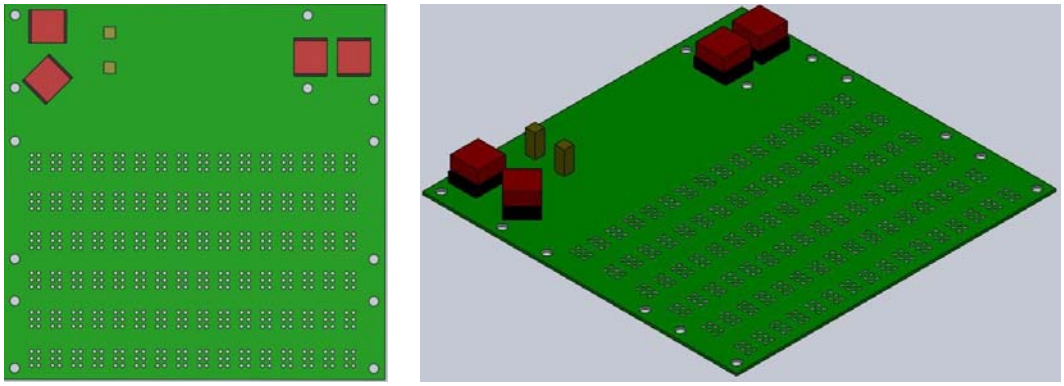


(a) Top view (b) 3D view

Figure 11: CAD screenshots of the Intermediate User Interface layout

2.3.3 Advanced user interface:

Braille students mastered the basic Braille concepts and the word creation, and who needs further practice in write multiple sentences can use this interface. In addition to that, this interface can be used as a digital Braille writer.



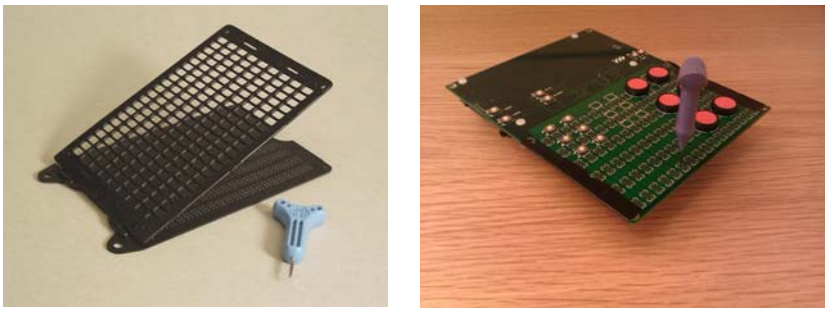
(a) Top view (b) 3D view

Figure 12: CAD screenshots of the Advance User Interface layout

2.4 Electronic Slate Implementation

There are two Braille cell implementations in this device: button based cell and the electronic slate (E-slate). As described earlier, the button based cell is intended for users learning the Braille concept and the e-slate is intended for advanced users. Figure **??** (a) shows a standard Braille slate and stylus,

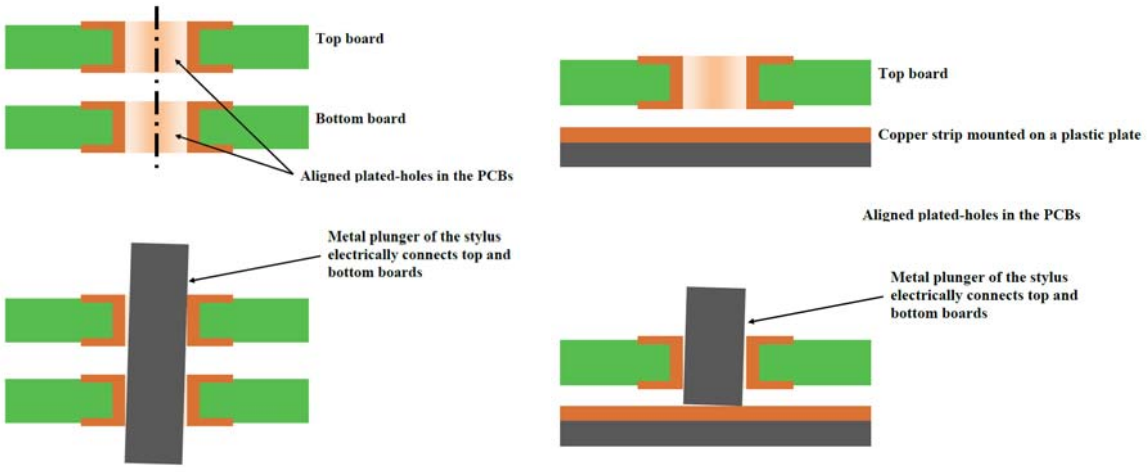
and Figure **??** (b) shows the implementation of the electronic slate in SABT. Here our primary intention is to use a standard stylus (with metal plunger) for writing Braille such that the user will have the same feeling when writing in a standard slate.



(a) Standard Slate and Stylus (b) E-slate and stylus

Figure 13: Standard slate for paper based Braille and Electronic slate in SABT

The concept is to use the metal plunger in the stylus to close an electrical circuit which will eventually send a signal to the microcontroller. The Figure **??** shows the implementation of the "circuit closing" with a stylus in both versions of SABT. In the version 1 of the SABT we used the approach which was already tested with the BWTv2 where the user need to pass the stylus through two aligned holes. Due the inherent nature of this design, the orientation of the stylus plays an important role in creating the electrical connection and ease of operation. In simple terms, the user need to insert the stylus perpendicular to the board in order to get the best outcome. Moreover, misalignments in the two boards can create significant issues during the operation. Fields trials provide us with valuable feedback on such problems and as a solution the new E-slate consist of only one set of holes and the plunger is connecting with a metal strip underneath the slate in order to close the electrical circuit.



(a) Version 1 (b) Version 2

Figure 14: Comparing E-slate hole design in two versions

Instantaneous feedback for user inputs require continuous monitoring every dot in the E-slate. One classic approach is to have digital output for each dot. However, due to large number of dots associated in this slate design, this approach impose several technical limitations in a low-budget system. Therefore we used the analog interface of the microcontorller to keep track of user interactions with the E-slate.

The bottom board consists of copper strips or the set of holes for each raw of the E-slate and these strips are interfaced with analog inputs of the microcontroller. E-slate uses a resistor network to provide each Braille dot in the top board of the slate with a unique voltage level as illustrated in the Figure **??**). For example, in our E-slate design we have sixteen Braille cells per raw which require 96 different voltage levels. Inserting the stylus to the Braille dot completes the circuit and generates a unique voltage at the analog input of the microcontorller. In the microcontorller software, this voltage level is compared with a pre-calibrated lookup table and interpret the Braille dot pressed.

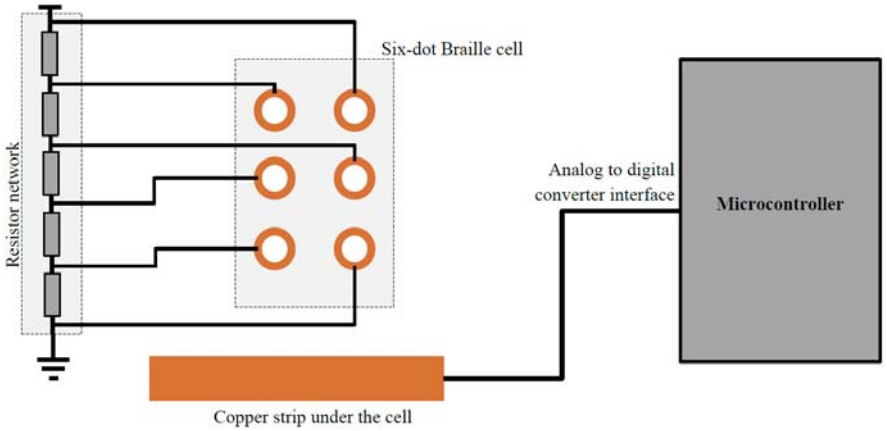


Figure 15: Resistor network provides unique voltage for each Braille cell

3 Electronic Design

This section explains the electronics development in SABT, primarily focusing on the design considerations and overview of each module in the design. The schematics and the PCB layouts of the design can be found in the appendix. First we consider the electronic design of the main control module.

3.1 Main Controller Module

3.1.1 Overview

This module essentially holds all the processing elements in the system. The Figure **??** illustrates the interactions of different elements with the embedded processor: an ATmega1284P low-power 8-bit microcontroller from Atmel devices. SABT is designed to operate with 4 AA batteries, thus the voltage level in the system is limited to 4.8V (typical AA rechargeable batteries are rated at 1.2V). This entails that component selection is limited to devices with 3.3V operational voltage. Here we did not consider using boost power supply to maintain the system voltage at 5V, due to elevated power consumption. ATmega1284P microcontroller consists of an internal oscillator which can provide clock frequencies in the 7.3-8.1MHz range. Moreover this can operate in 2.7-5.5 V voltage range with the selected 8MHz calibrated internal clock frequency.

The ATmega processor in the main control module basically has two task sets: executing driver level tasks and execute higher level tasks related to processing user interactions. The former tasks are essential for proper operation of the electronics and the latter is based on the commands from the computer and the user interface. In this section we only consider the driver level tasks and related electronics designs. The higher level software implementation details are discussed in the next section of this report.

Driver level tasks of the controller:

1. Communicate with the user interface module
2. Communicate with an external computer via USB
3. Playback audio files via audio codec

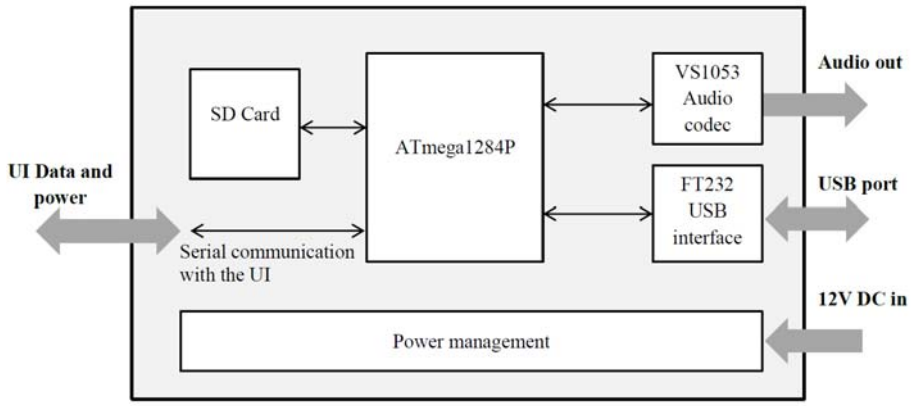
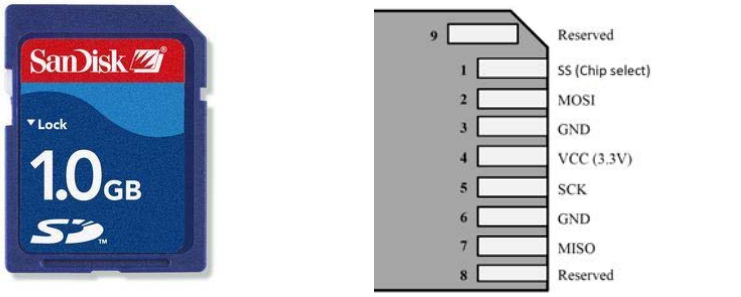


Figure 16: Block diagram representation of the Main Control Module

4. Read and Write SD card to store user data

3.1.2 SD card interface

In order to make accessible for visually impaired users, SABT has an audio feedback method for every user action with the device. This entail large storage for keeping required audio files. Considering cost, availability, maintainability issues we selected SD card based storage system. Secure Digital (SD) cards can be found in various storage capacities, which enable the users to purchase appropriate storage space for their requirements. Figure **??** shows an external view of a 1GB SD card and the pins used in the SPI bus interface of the SD Card.



(a) SD card (b) Pin configuration for SPI interface

Figure 17: Secure Digital (SD) card and the SPI pin layout

SD card has two communication interfaces: SD bus interface and SPI interface. SD bus is a 4-bit parallel data interface which is faster than the serial SPI interface. However, in this design we used the SPI based communication to interface SD card with the microcontroller primarily due to implementation considerations of the communication interface. The ATmega1284P microcontorller consists of a in-built SPI interface which we can utilize for communicating with external peripherals. In this design we used the built-in SPI to communicate with three external interfaces: SD, MP3 data and MP3 control. This will be further explained later in this section.

3.1.3 VS1053 MP3 Interface

The audio feedback interface of SABT is developed around a MP3 decoder chip from VLSI systems. The VS1053 audio decoder is a single-chip multiple audio format decoder (Ogg Vorbis/MP3/AAC/WMA and MIDI) with streaming support for MP3 and WMV. Also contains in-built high-quality variable-sample rate stereo ADC and stereo DAC together with an earphone amplifier. Inclusion of the DAC, the earphone amplifier and the ability to operate in 3.3V makes this the ideal candidate for SABT, since

it significantly reduces the complexity, development effort and time of the device. Figure **??** represents a simplified block diagram of the VS1053 chip, which highlights the features/modules we use in this design.

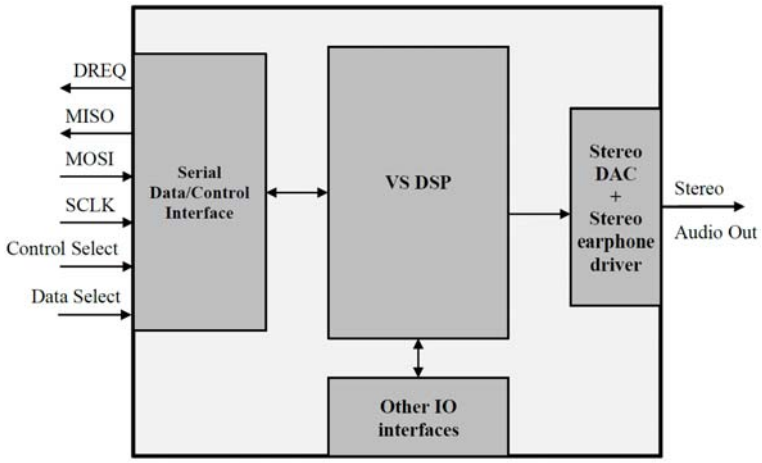


Figure 18: Simplified block diagram of the VS1053 Audio decoder IC

The VS DSP unit is the core of this chip which processes the analog and digital signals and produces appropriate output signals. Digital audio data and control signals are interfaced with the DSP via a SPI bus. The control interface is to provide commands for the MP3 decoder and the data interface is to stream audio data in the chip. The analog and digital sections of the IC require a separate power supplies which make the analog audio output free from digital noise. For more details on the schematic and electronic implementation please refer Figure **??** in the appendix.

3.1.4 USB Interface

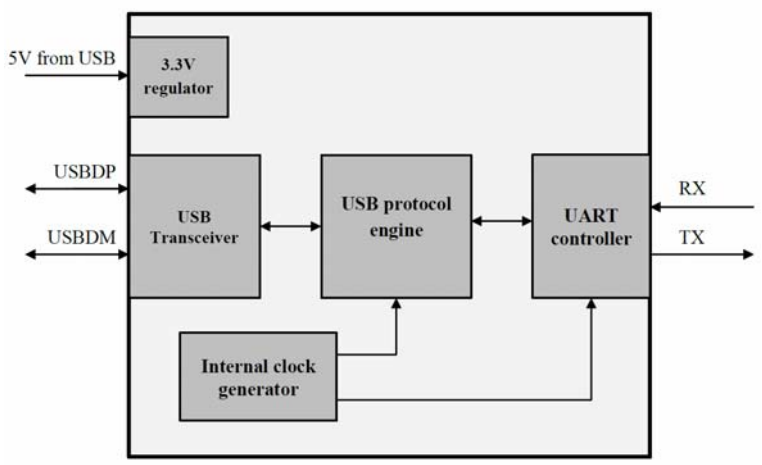


Figure 19: Simplified block diagram of the FT232 USB-Serial converter IC

The ATmega microcontorller in the SABT main control module have three communication modules: SPI, UART and IIC. Among them the UART (Universal Asynchronous Receiver/Transmitter) is the primary interface intended to communicate with external devices. In other words; the SPI and IIC interfaces are more suitable for communication with the devices in the same circuitry and UART interface is designed to communicate with external devices in full-duplex mode without clock synchronization. ATmega1284P device consists of two USART modules: we are using one for the communication with the UI module and the other is used to communicate with external device using USB protocol. The

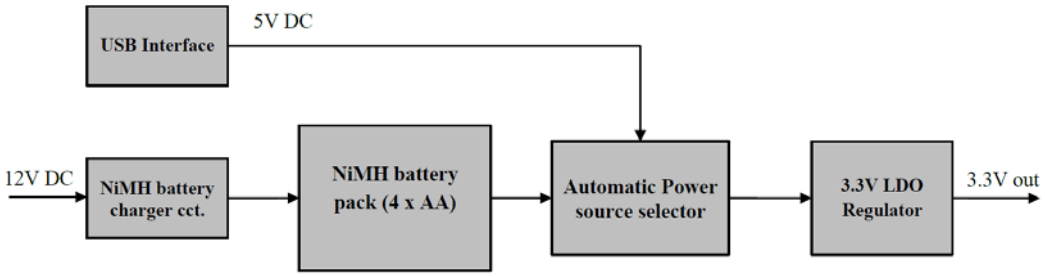


Figure 20: Block diagram representation of the power management sub-system

other common alternative for USB is the RS232 serial interface which is becoming obsolete in personal computers. Instead USB interface gained a lot of popularity in personal computers over the past decade as the communication interface with external devices.

Since the UART in ATmega do not have a USB compatible signals or relevant communication protocol implementation, we are using an external TTL serial to USB converter IC: FT232LT. A simplified block diagram of this converter IC is illustrated in the Figure **??**. Apart from using as the communication interface with a computer, we are using the 5V power from the USB port as an alternative power source for the SABT when it is connected to a computer. More details on the power management is provided in the next section.

3.1.5 Power management

SABT is designed to operate with 4 rechargeable AA NiMH batteries, as the primary power source. Availability, cost and long-term support for AA batteries, as well as easy replacement with more commonly available AA alkaline batteries are the main reasons behind this selection. The batteries are charged with an in-built NiMH charger circuit which can be powered from any external 12V DC power supply. Moreover, this enable the users to power the device with a 12V DC wall adapter when used with rechargeable batteries. Apart from the primary power source, SABT can operate with USB power when it is connected to a computer. The power management system automatically switches to USB power once connected to a computer. Figure **??** illustrates the block diagram representation of the power management system. Refer Figure **??** in Appendix for the schematic design of the power management system. As mentioned earlier in this section, NiMH AA batteries are rated at 1.2V when fully charged, and the system is designed to operate with 3.3V supply. We used TPS79633DCQ low-dropout (LDO) low-power linear voltage regulator [**?**] which enable our system to operate even with 3.6V input which corresponds to a fully discharged NiMH battery [**?**]. Switching between battery and USB power sources is managed by LT4412 Low Loss PowerPath Controller from Linear Technology [**?**]. Since the USB supply voltage (5V DC) is always higher than the battery pack supply voltage (4.8V), the system will always switch to USB powered configuration when connected with a computer. MAX713 NiMH fast-charge controller IC

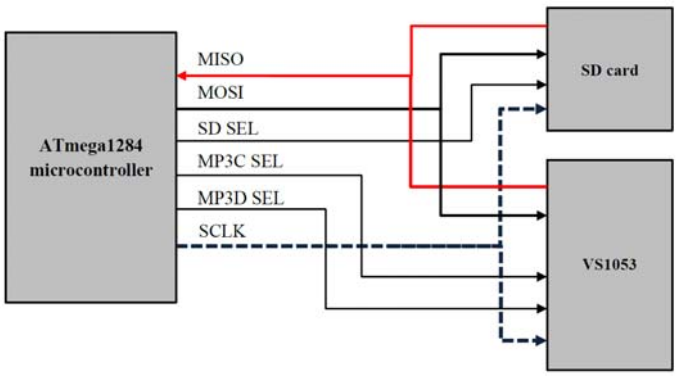


Figure 21: Microcontroller connections with SPI devices

is used for managing AA NiMH battery pack.

3.1.6 SPI communication for SD, MP3 data and MP3 command interfaces

Three external interfaces are connected with the SPI bus of the microcontroller: SD card, MP3 data interface and MP3 control interface. The ATmega1284P acting as the SPI master, uses device select feature to select the appropriate device for communication as illustrated in Figure **??**.

The SPI interface is primarily used to its full extent in the MP3 playback process, where the microcontroller streams MP3 data from the SD card to the VS1053 decoder IC. The FAT32 file structure in the SD card stores information in 512 byte sectors and the MP3 decoder accepts data in 32 byte blocks. The following psudo code segment represents the design of the MP3 playback function. Note that we are always sending 32 byte segments to the audio decoder without checking the status of the DATA_REQUEST pin, this will ensure smooth playback of the audio file. Moreover, after each audio decoder write, the code checks for user inputs from the UI module and the computer. This enable to users to interact with the system while audio playback operation.

Psudo representation of the MP3 streaming code

```
MP3_play()
{
 LocateMP3_FileInSD_Card();
 while(!end_of_fi1e)
 {
  get_Next_Cluster()
  for(each sector in that cluster)
  {
    bytes_in_block=ReadSingleBlock();
    while(i<bytes_in_block)
    {
     if(DATA_REQUEST)
     {
      Send_32AudioBytesToMP3decoder();
      i+=32;
     }
     Look_for_user_inputs();
    }
  }
 }
}
```

3.2 User interface module

In this section we study the electronics and the software in the user interface module. Earlier in this report we studied that there are three user interface types designed to accommodate users of different skills. However, the underlying electronics and processing in all three modules are similar. The user interface module has a 8-bit low-power ATmega microcontroller ATmega168PA which interprets the user interactions and report them to the main control module. All the components in the UI boards and powered from 3.3V DC supply. The connection with the main control module provides power to the UI board, and provides the interface for data communiation.

The user interactions are sensed via two major interfaces in the microcontroller: digital and analog intefaces. All the control inputs are interfaced with digital input pins of the microcontroller. A typical push buttton input is illustrated in the Figure **??**. In addition to those, the six-button braille cell in the primary user interface is configured to use the digital interface. This was possible since those six-button are the only Braille inputs in the primary interface (see Figure **??**) The large number of Braille inputs associated with other user-interface types makes it technically challenging to configure them as digital inputs.

In the intermediate and advanced user interfaces, the Braille inputs are monitored using the analog channels of the microcontroller. As explained in section **??**, the intermediate UI has two Braillcell

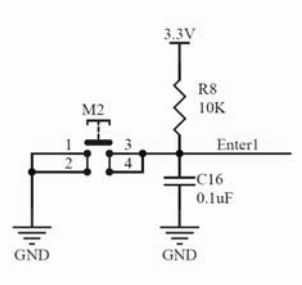


Figure 22: Digital signals in the user interface module

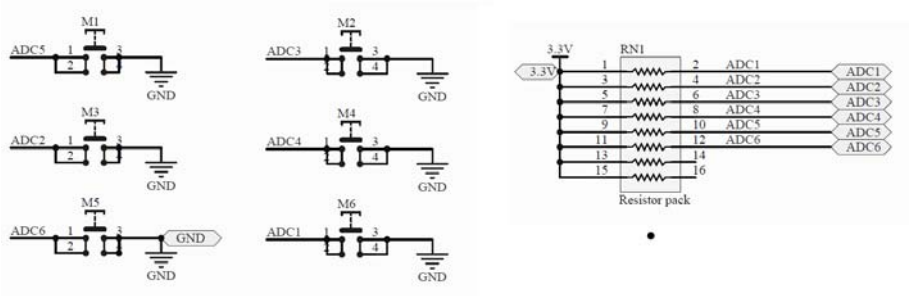


Figure 23: Braille cell implementation in the primary user interface. In this interface the microcontorller ADC pins are configured as digital inputs.

implementations: button-based Braille cells and 2 raw E-slate whereas the advanced UI consists of a 6 raw E-slate. More details on the E-slate implementation is presented in the section **??**.

Unlike the digital input monitoring, the inherent nature of the analog signals and given the large number of inputs associated with E-slate interface, the analog signals need thorough software filtering and sanity checks. In SABT, single analog channel in the ATmega16 microcontorller is monitoring a single E-slate row, which corresponds to 96 analog signal levels. During the rise time of the analog signal (i.e. from 0 voltage to V_{dot}) it moves through several calibrated voltage levels in the calibration table. This might causes several wrong identifications along the path (see Figure **??**), if correct precautions were not taken.

In this design, we have implemented the following filtering algorithm:
If $\|\Delta V(t)\| < \Delta V_{TH}, \forall t = [T_n, T_{n-1}, ..., T_0]$ and $V(T_0) > 0$, then the system recognizes the current voltage level as a valid Braille dot. Here, $\Delta V(t) = V(t) - V(t-1)$ and ΔV_{TH} is the threshold value for voltage fluctuation inside a single calibrated dot. This is determined by the number of voltage input levels for a particular analog input line and the measurement resolution of the analog port. In this design, we have defined $\Delta V_{TH} = \dfrac{2^N - 1}{2n}$, where N and n are the ADC resolution in number-of-bits (i.e. 8-bit, 10-bit) and number of inputs per analog pin respectively. In above expressions, $V(t)$ is the voltage level in the analog pin at time t and $T_i = T_0 - i \times \Delta T$, where T_0 is the current time and ΔT is the timer interval used for this filter.

3.3 Communications

SABT performs two types of device interactions: in-circuit device communications where the communication protocol is pre-defined by the device manufacturer, and the inter-module/computer communications where we use custom communication protocols. For instance, the interactions with SD card and audio codec via the SPI bus uses the communication protocols defined by the SD controller and the VS1053 chip. In contrast, for the interactions with User Interface and external computers, SABT uses its own communication protocols as describes in this section.

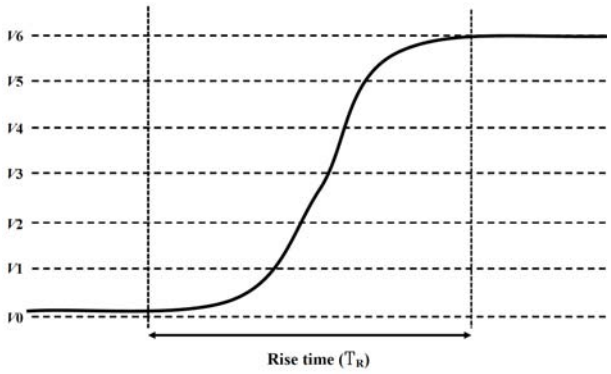


Figure 24: Analog signal variation and triggering detecting wrong input dots. In this example, the analog signal passes five calibrated input levels during the rise-time.

Table 1: Message structure used in inter-module communications

Message element	Description

Prefix
`[U][I]`
`[M][C]`
	The UI module sends the two-byte prefix ASCII "UI" to indicate the sender of the message and Main control module sends the message with two-byte pre-fix "PC"

Message Length `[msglen]`	Unsigned 8-bit integer to indicate the length of the payload
Message Number `[msg_number]`	Unsigned 8-bit integer to keep track on messages. This is helpful to track responses for requests made by other modules.
Message Type `[msg_type]`	8-bit message type identifier. This identifier is used to decide the byte sequence and their meaning in the payload. More details of this is provided in the Table **??**
Payload `[payload]`	Message payload of the size defined by message length. More details of this is provided in the Table **??**
16-bit Checksum `[CRC1][CRC2]`	Two bytes storing the 16-bit checksum of the data packet. Prefix, message length and CRC bytes are not included in the checksum calculation.

3.3.1 Inter-module communications

The messages send by the User interface to the Main control module has the following format. Each element in the message structure is explained in the Table **??** and **??**.

`[U][I][msglen][msg_number][msgtype][payload][CRC1][CRC2]`

`[M][C][msglen][msg_number][msgtype][payload][CRC1][CRC2]`

In the main control module, data from the UI modle is monitored as listed in Listing 1. Once the data packet received completely, the UI_parse_message function in UI_Handle.c is called which interprets the payload content and send appropriate command to the mode. The complete UI_Handle.c code in provided in the Appendix.

3.3.2 Computer communications

Unlike UI communications, SABT-PC communication protocol is designed to enable the users to interact with the device via any terminal program such as windows hyper-terminal, PuTTY or minicom.

`[prefix][msgtype][payload][LF][CR]`

A simple message structure is used which use human readable ASCII characters (i.e. numbers and text) and the end-of-message is marked with a Line feed, Carriage Return combination. A two byte prefix "PC" is used to identify messages originating from PC and four-byte "SABT" prefix is used to identify messages originating from the SABT.
Message Type `[msgtype]`: The current version of the SABT is accepting two message types: initialization message and the mode modify message.
Payload `[payload]`: Contains the message corresponding to the `msgtype`.

Initialization message - ASCII 'x'
Init message does not contain any payload. SABT uses this message for providing auto-detect support for PC software. Upon reception of init message SABT sends an acknowledgement message `SABT-v2.1`. Here `v2.1` is the hardware/firmware version of the device.

Modify modes file - ASCII 'M'
Payload of this message type contains the numbers of the selected modes for activation. Upon successful completion of SD card writing SABT sends `SABT-OK` message to the computer. If the SD card writing operation did not completed SABT send `SABT-FAIL` message. Users can either use terminal application or the mode manager utility to perform this operation. Payload of this message will look similar to

Table 2: Message types used for inter-module communications

Message type (ASCII value)	Payload content
65	Payload contains a Braille dot and its location in the UI payload[0] - Braille dot value (1-6) payload[1][2] - Braille cell position (row,column) Note 1 - This is valid only for UI to MC communications Note 2 - Primary interface does not send the Braille cell location Note 3 - Button based cells in the intermediate interface is considered as the first row
66	Payload contains a Braille character and its location in the UI payload[0] - Braille cell value (1-64) payload[1][2] - Braille cell position (row,column) Note 1 - This is valid only for UI to MC communications Note 2 - Primary interface does not send the Braille cell location Note 3 - Button based cells in the intermediate interface is considered as the first row Note 4 - This is sent after the UI receives "End-of-character" message (i.e. "Enter" button pressed)
67	Payload contains an Error message.
68	Payload contains a control button input from UI. payload[0] contains values 1 to 6 and, represent command buttons Enter, Exit, Mode Forward, Mode Backward, Volume Up and Volume Down respectively Note 1 - This is valid only for UI to MC communications
69	Miscellaneous This message type can be used to pass non-standard messages between the modules.

Listing 1 - User Interface message receive code

```
void USART_Keypad_ReceiveAction(void)
{
  USART_Keypad_DATA_RDY=false;

  if(!USART_UI_header_received)
  {
    USART_UI_prefix[2]=USART_Keypad_Received_Data;
    USART_UI_prefix[0]=USART_UI_prefix[1];
    USART_UI_prefix[1]=USART_UI_prefix[2];
    if((USART_UI_prefix[0]=='U')&&(USART_UI_prefix[1]=='I'))
    {
      USART_UI_header_received=true;
      USART_UI_ReceivedPacket[0]=USART_UI_prefix[0];
      USART_UI_ReceivedPacket[1]=USART_UI_prefix[1];
      USART_UI_receive_msgcnt=2;
      USART_UI_length_reveived=false;
    }
  }
  else if(!USART_UI_length_reveived)
  {
    if(USART_UI_receive_msgcnt==2)
    {
      USART_UI_received_playload_len=USART_Keypad_Received_Data;
      USART_UI_ReceivedPacket[USART_UI_receive_msgcnt]=USART_Keypad_Received_Data;
      USART_UI_length_reveived=true;
      USART_UI_receive_msgcnt++;
    }
    else
    {
    USART_UI_header_received=false;
    }
  }
  else
  {
    USART_UI_ReceivedPacket[USART_UI_receive_msgcnt++]=USART_Keypad_Received_Data;
    if(USART_UI_receive_msgcnt==USART_UI_received_playload_len)
    //full message has been received
    {
      USART_UI_Message_ready=true;
      USART_UI_header_received=false;
      USART_UI_length_reveived=false;
    }
  }
}
```

<1><3><15>$ that contain the list of modes to activate. More details on the mode management and mode manager utility is provided in the next section of the report.

Listing 2 - PC message receive code

```
unsigned char USART_PC_ReceiveAction(void)
{
    USART_PC_DATA_RDY=false;
    if(!USART_PC_header_received)
    {
      USART_PC_prefix[2]=USART_PC_Received_Data;
      USART_PC_prefix[0]=USART_PC_prefix[1];
      USART_PC_prefix[1]=USART_PC_prefix[2];
      if((USART_PC_prefix[0]=='P')&&(USART_PC_prefix[1]=='C'))
      {
        USART_PC_header_received=true;
        USART_PC_ReceivedPacket[0]=USART_PC_prefix[0];
        USART_PC_ReceivedPacket[1]=USART_PC_prefix[1];
        USART_PC_receive_msgcnt=2;
      }
    }
    else
    {
      if(USART_PC_Received_Data==13)
      //If carraige return found --> end of the command
      {
        USART_PC_received_playload_len=USART_PC_receive_msgcnt;
        USART_PC_Message_ready=true;
        USART_PC_header_received=false;
      }
      USART_PC_ReceivedPacket[USART_PC_receive_msgcnt++]=USART_PC_Received_Data;
    }
    return 0;
}
```

4 Modes

As mentioned earlier in this report, SABT firmware has two major tasks; driver level tasks and higher level task which implement the basic functionality of the Braille tutor. Majority of the higher level tasks are associated with modes: navigating modes and mode implementations. SABT can accommodate multiple user-interaction programs aimed at different learning aspects. These user-interaction programs are called modes. In SABT, all the modes follow a template such that they are compatible with the other software elements interacting with modes. Apart from that, using a template enables multiple developers to develop modes, which can be compiled together using the mode manager software utility. In this section, we will explore the following: anatomy of a mode (based on the mode template), interaction with modes with user interface commands and the mode selection process.

4.1 Anatomy of a mode

Before we discuss the anatomy of a mode in detail, we first have a quick look at how the ModeManager[1] works. ModeManager is a cross-platform software utility written in java. The primary intention of this software is to enable multiple developers to introduce modes to the SABT and able to compile the modes without conflicts. In order to achieve that, modes are developed using a template as defined in this section. Upon starting the ModeManage software, the user needs provide the path to the folder that contain all the modes. Then the software searches every folder in that path for mode identifier files, which contain all the details of the mode in that particular folder. The ModeManager assumes that all the files required for that mode are in that same folder.

A mode consists of several files: files for mode code, mode identifier files, and sound files for the mode. The code consists source files (`.c` and `.h`), which we discuss in detail later in this section.

The mode identifier file `MODE.mod` provides the name, description and the compatible UI types for the mode in the format listed here (MODE.mod).

In addition to that every mode must have the following sound files:
`MD$XX$.MP3`: this file is played when the users are navigating through the main menu. This should say the name of the mode.

[1]ModeManager User's guide is provided in the Appendix

Listing - MODE.mod

```
#name name of the mode
#desc description of the mode
#Interface_Type
#Interface_Type
```

`MD$XX$_WC.MP3`: this file is played once a user selects the mode. This should be a welcome message to that mode and give the users a brief description on how the mode works.

Moreover, a mode is typically associated with number of other media files (.MP3 format) which provides sounds for that mode. Files in all the user created modes should be named as `MD$XX$_identifier.MP3`, and the file should be addressed by the same name within the mode.

The source and header files of the mode are named as: `MD$XX$.c` and `MD$XX$.h`. Note that this naming convention is a requirement of the ModeManager software. The example mode file listed in Appendix 3 provides more information on the source code formatting. A mode, at least, should contain following functions:

```
void MD$XX$_Main(void)
void MD$XX$_Reset(void)
void MD$XX$_CallModeYesAnswer(void)
void MD$XX$_CallModeNoAnswer(void)
void MD$XX$_InputDot(char thisDot, int iRow, int iCol)
void MD$XX$_InputCell(char thisCell, int iRow, int iCol)
```

UI_handle software module calls these functions in response to the user inputs via UI modules. In addition to those, a mode can contain unlimited local functions to implement the mode functionality. The "Dot Practice" example in appendix provides implementation of such function.

Note that, in order to comply with ModeManager specifications, all the mode functions and variables should start with the "`$MDXX$_`" term. This will enable the ModeManager to compile the source without conflicts with other modes.

Now we shall explore the compulsory functions in a mode. All the above functions are events that are triggered by the `UI_Handle` module. `UI_Handle` module keeps track on all the inputs for the UI module and takes appropriate action for the events. More details on this module is provided in later in this section and the source code is listed in Appendix 3.

Main()
Once a mode is selected, this routine is executed continuously by the SABT core, i.e. it is added to the main loop of the SABT core. Note that an infinite loop within this function will halt the operation of SABT until a manual reset. The user code should quickly process the information and return from the main function.

Reset()
Upon selection of a mode, this function is called once by the `UI_Handle` module. This should be considered as a constructor for that mode, and all the initialization code should be placed here.

CallModeYesAnswer() and CallModeNoAnswer()
These two functions are called by the "Enter" and "Escape" button presses, once in the mode. A mode can utilize these events to get feedback from the users, as to "YES" or "NO" answer to a question.

InputDot(char thisDot, int iRow, int iCol)
This event is triggered once the UI_Handle detects a dot input, i.e. the user has pressed a button in a Braille cell or used the stylus to mark a dot.

InputCell(char thisCell, int iRow, int iCol)
This event is triggered once the UI_Handle detects a cell input, i.e. the user has pressed the "Enter" button after completing a letter in a Braille cell or in the slate. In addition to the input letter, this function provides the location of the Braille cell in the slate.

Playing a sound file
Playing a sound file is handled by the core SABT code, in which a series of coordinated tasks are executed according to the manufacturer's specifications of the audio decoder and the SD card (see section **??** for more details). Any request to play an audio file must come through the RequestToPlayMP3file("filenametobeplayed") command. This request is then processed by the audio decoder module, once the current playback operation - if any, is completed.

4.2 Mode management

As explained in the ModeManager User's guide, a selected set of modes can be programmed into the SABT using ModeManager. These modes may be used by multiple user-interfaces or are restricted to certain user-interface. A fair amount of technical know-how is needed to program the binary file created by ModeManager. Therefore we have developed another approach for selecting appropriate modes to be used with current user-interface as well as the current learning objectives, which can be easily used by ordinary users. The "mode selector" feature embedded into the SABT-core firmware is providing this functionality using a simple configuration file placed in the SD card.

4.2.1 Mode handling inside the firmware

The UI_Handle software component handles all the interpretations of the user inputs and call relevant functions in the modes. Figure **??** shows how modes interact with the SABT core via UI_Handle module. Apart from providing the interface to modes, this module loads the selected modes into the memory as explained below. Note that complete listing of the UI_Handle module is provided in the Appendix.

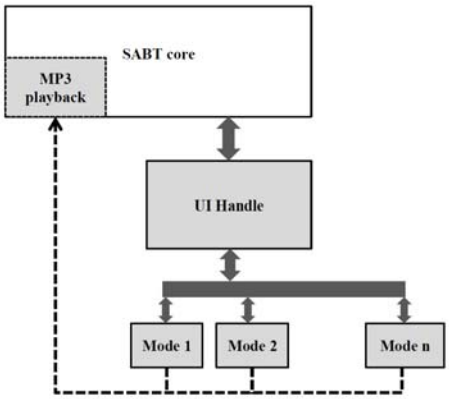


Figure 25: Interaction of modes with different modules in SABT firmware

The SD card contains the mode selection configuration file (`modes.dat`) which defines the selected the modes as well as the order in which they appear in the mode menu of the SABT. The `modes.dat` file contain the selected modes as in the following example:
`<1><3><15>$`
In the example above three modes are selected 1,3 and 15. Every mode number is enclosed with `<` and `>` brackets. The statement is ending with `$` sign. Having the mode set as `<1><15><3>$` selects the same set of modes but they are ordered in different manner.

Upon start, SABT firmware looks for the `modes.dat` configuration file in the root area of the SD card. Upon successful detection of the file, it loads the selected modes into memory as listed in the following code segment (Listing 3).

4.2.2 Changing mode file

The `modes.dat` file can be altered in several approaches: copying a new file to the SD card, using terminal program and using the mode selector software.

Listing 3 - Load selected modes from SD card

```
bool UI_CheckModes(void){
  unsigned char FileContent[100];
  unsigned char ModeID[3];
  int i=0;
  int iMoN;
  bool bBoNFound;
  const char* ModesFile="MODES.DAT";
  Number_of_modes=0; //This variable is global (int type)
  for(i=0;i<100;i++)
    FileContent[i]=0;
  if(readAndRetreiveFileContents (ModesFile,FileContent)>0)
    return false;
  bBoNFound=false;
  i=0;
  while(FileContent[i]!='$'){
    if(FileContent[i]=='>')
    {
      UI_Modes[Number_of_modes]=atoi(ModeID);
      Number_of_modes++;
      bBoNFound=false;
    }
    if(!bBoNFound)
    {
      ModeID[0]=0;
      ModeID[1]=0;
      ModeID[2]=0;
      iMoN=0;
    }
    else
    {
      if(iMoN==3) return false;
      ModeID[iMoN++]=FileContent[i];
    }
    if(FileContent[i]=='<') bBoNFound=true;
    i++;
  }
  return true;
}
```

Direct copying : In the first method, the SD card need to be removed from the system and connected to a computer SD card reader to access the `modes.dat` file. Once the new file is in place, insert the SD card to the SABT unit and switch on the device.

Using Terminal Emulator Software : In the second method, the user does not need to remove the SD card and manually copy the new file. Instead users can use the USB port to communicate with SABT using any commonly available terminal emulator programs such as windows hyter-terminal, PuTTY, minicom, etc. Here terminal based communication is used to modify the `modes.dat` as explained below:

PC command	**SABT reply**	**Description**
`PCx`	`SABT-v2.1`	In order to make sure SABT device is connected to this port, send the self-identify command SABT replies with the version
`PCM<1><6><2><7>$`	successful: `SABT-OK` unsuccessful: `SABT-FAIL`	Then send the mode selection command As an example, we are selecting modes 1,6,2 and 7 in that order Upon successful completion of the process, power toggle the device will load the newly selected modes. In an event of the unsuccessful message, make sure the SD card write-protection is off and SD card is not full.

Mode selector software : The terminal based mode selection does not perform any error checks on the given list of modes, it simply replace the file content with the given string. Moreover, using direct copy

method and the terminal window method, the users need to have a clear idea on the modes programmed in the SABT and the order in which they are programmed. In order to perform an error free mode selection process, users are encouraged to use the mode selector software utility. Mode selector user's guide is provided in the appendix.

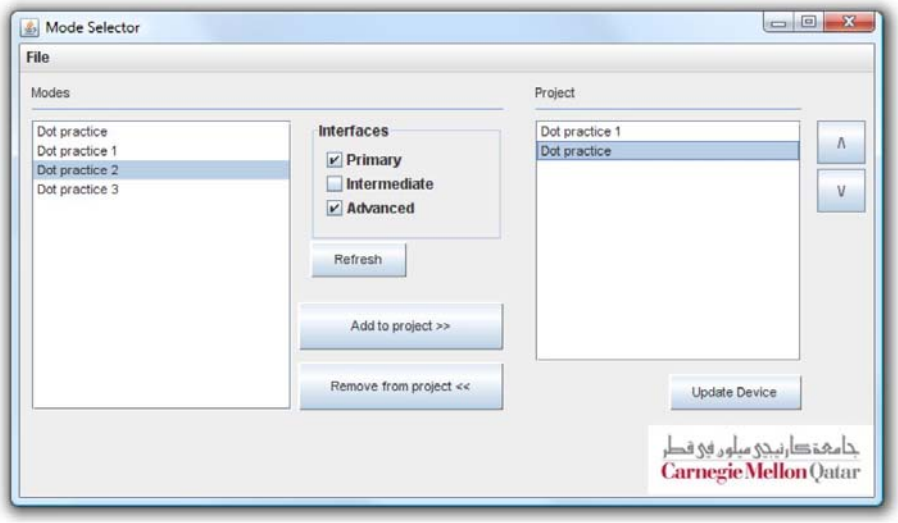


Figure 26: Mode selector graphical user interface provides the users with a convenient and error-free approach to select and sort required modes in the SABT

5 Future Directions

References

[1] M. Bernardine Dias, M. Freddie Dias, Sarah Belousov, Mohammed Kaleemur Rahman, Saurabh Sanghvi, and Noura El-Moughny. Enhancing an automated braille writing tutor. In *IEEE/RSJ International Conference on Intelligent RObots and Systems (IROS 2009)*.

[2] Nidhi Kalra, Tom Lauwers, and MBernardine Dias. A braille writing tutor to combat illiteracy in developing communities. In *AI in ICT for Development Workshop, International Joint Conference on Artificial Intelligence*, January 2007.

[3] Nidhi Kalra, Tom Lauwers, Daniel Dewey, Thomas Stepleton, and MBernardine Dias. Iterative design of a braille writing tutor to combat illiteracy. In *Proceedings of the 2nd IEEE/ACM International Conference on Information and Communication Technologies and Development*, December 2007.

[4] Rotimi Abimbola, Hatem Alismail, SarahM Belousov, Beatrice Dias, MalcolmFrederick Dias, MBernardine Dias, Imran Fanaswala, Bradley Hall, Daniel Nuffer, ErmineA Teves, Jessica Thurston, and Anthony Velázquez. istep tanzania 2009: Inaugural experience. Technical Report CMU-RI-TR-09-33, Robotics Institute, Pittsburgh, PA, August 2009.

[5] M Bernardine Dias, Malcolm Frederick Dias, SarahM Belousov, Mohammed Kaleemur Rahman, Saurabh Sanghvi, Fun-Factor Noura El-Moughny", title = "Enhanced Formalization, and Field Testing for a Low-Cost Braille Writing Tutor. Technical Report CMU-RI-TR-09-30, Robotics Institute, Pittsburgh, PA, July 2009.

[6] SarahM Belousov, Yonina Cooper, MBernardine Dias, MalcolmFrederick Dias, Jen Horwitz, Brian Manalastas, Jonathan Muller, Aysha Siddique, Anthony Velazquez, and ErmineA Teves. istep 2010 bangladesh. Technical Report CMU-RI-TR-35, Robotics Institute, Pittsburgh, PA, May 2011.

[7] Texas Instruments Incorporated. TPS79633DCQ datasheet - Ultralow-Noise, High PSRR, Fast, RF, 1A, Low-Dropout Linear Regulators (Rev. N).

[8] Inc. Energizer Holdings. Energizer NH15-2300mAh datasheet, EBC - 7102WB.

[9] Linear Technology Corporation. LT4412 datasheet - Low Loss PowerPathTM Controller in ThinSOT.

A Appendix - Schematic designs

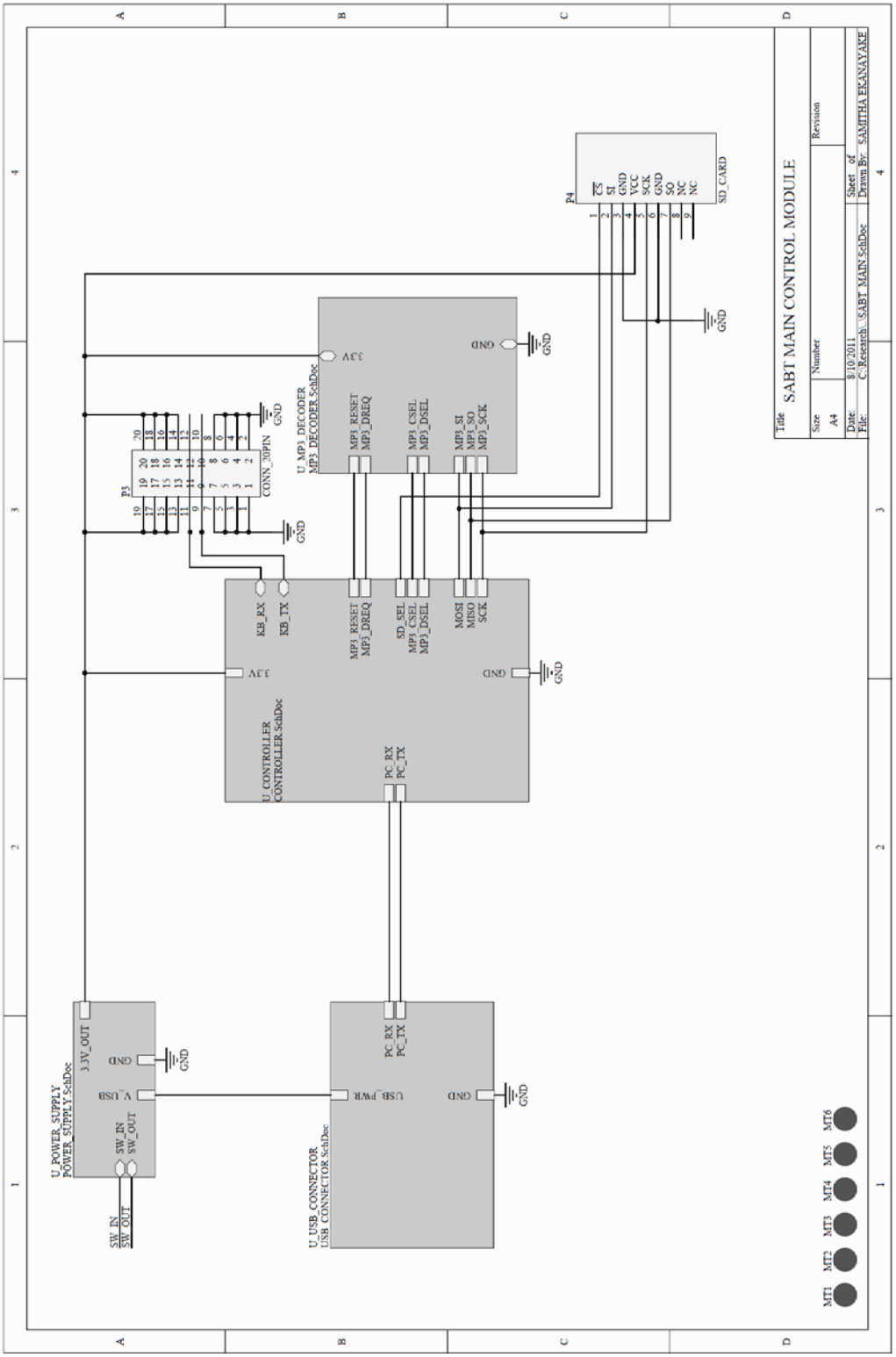


Figure 27: Interconnection of different sub-modules in the main control module of the SABT. The connection to the SD card (P4) and the user interface connector (P3) are shown here.

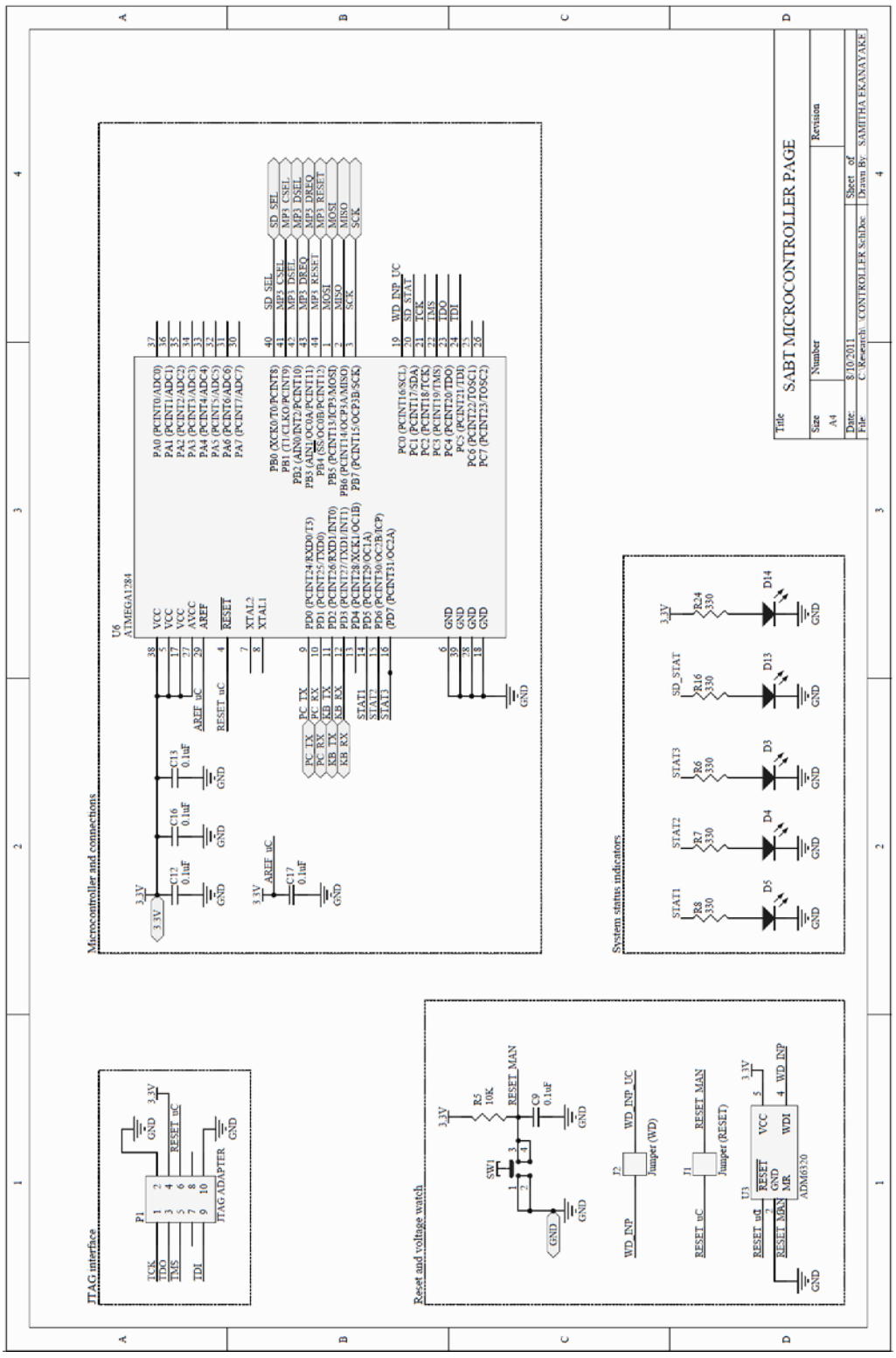


Figure 28: ATmega1284P microcontroller and basic connections to it. Main control module has four customs LED status indicators as shown in the schematic. One of them was dedicated as a SD card status indicator and other three can be user for debugging information. These indicators are primarily intended for developers and technicians debugging the device.

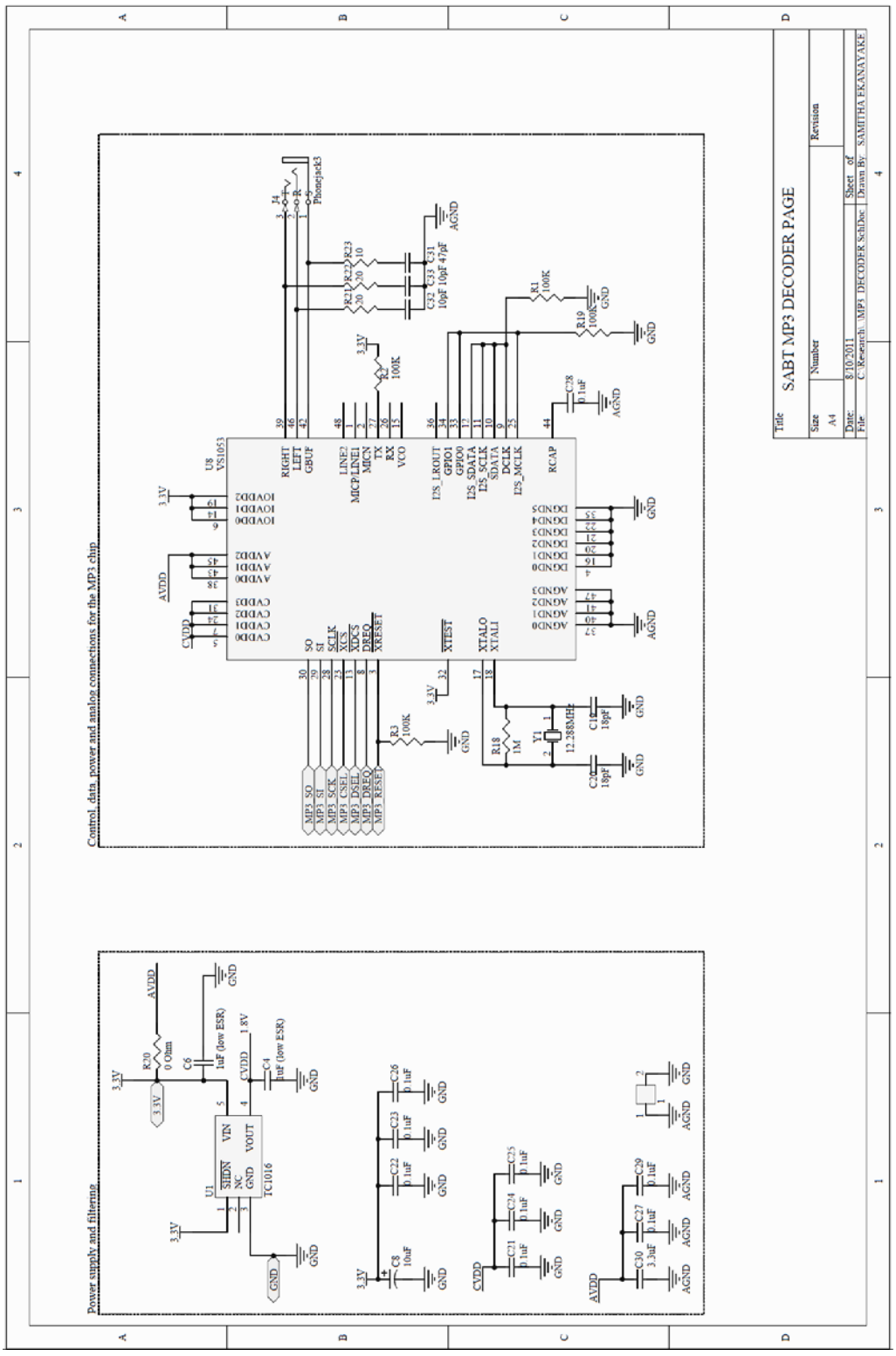


Figure 29: VS1053 audio decoder sub-module schematic. This consists of a 1.8V voltage regulator for powering the digital/control interface of the audio chip. In order to minimize digital switching noise in the analog audio circuitry we have used separate analog ground copper pour and connected it to the common ground plane at a single point.

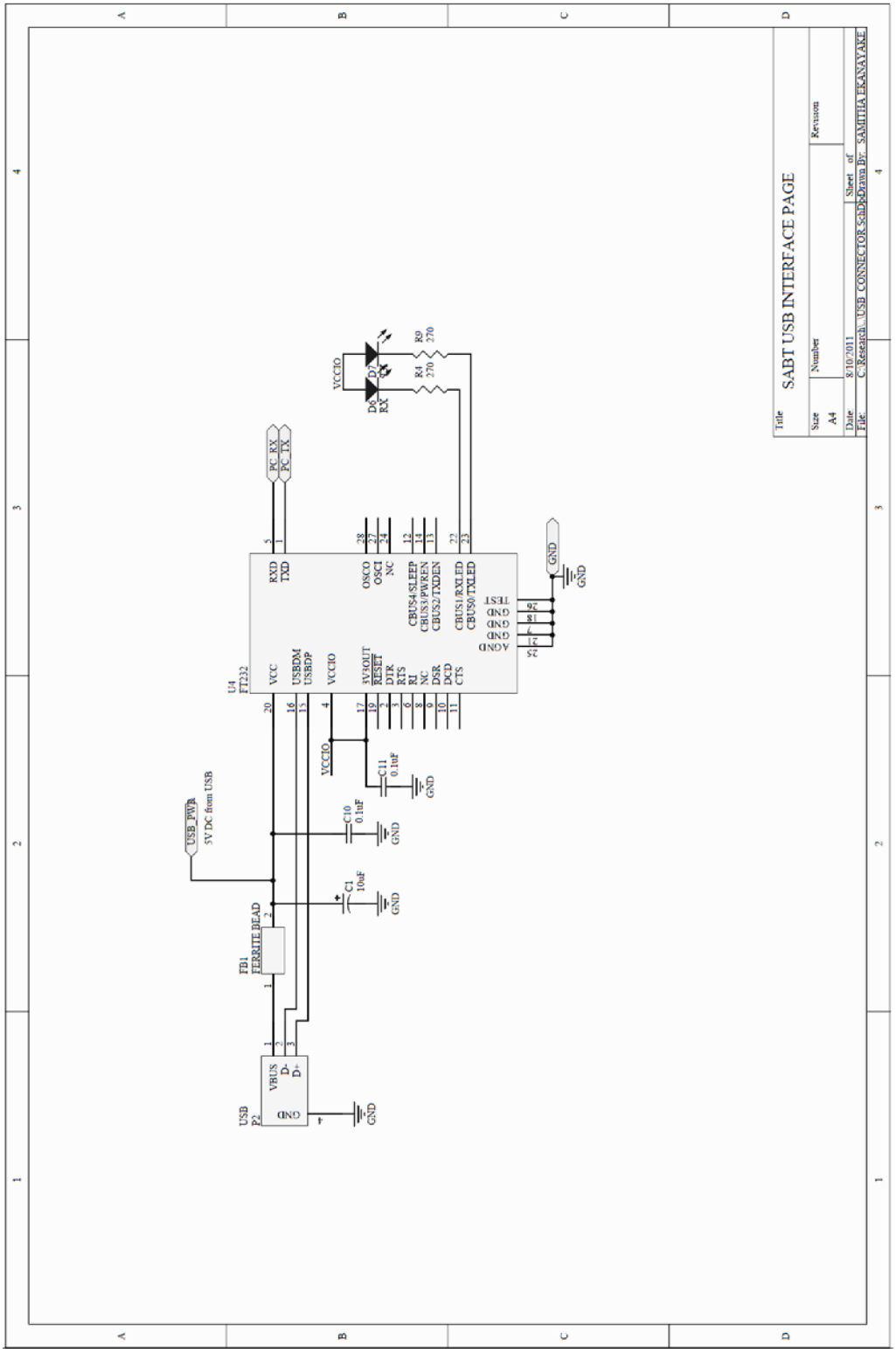


Figure 30: FT232 based USB-Serial converter sub-module schematic. Note that we have included two status LED to monitor TX/RX communications with computer. Also note the 5V out line drawn from the USB port. This USB supply is powering the device once it is connected to a computer.

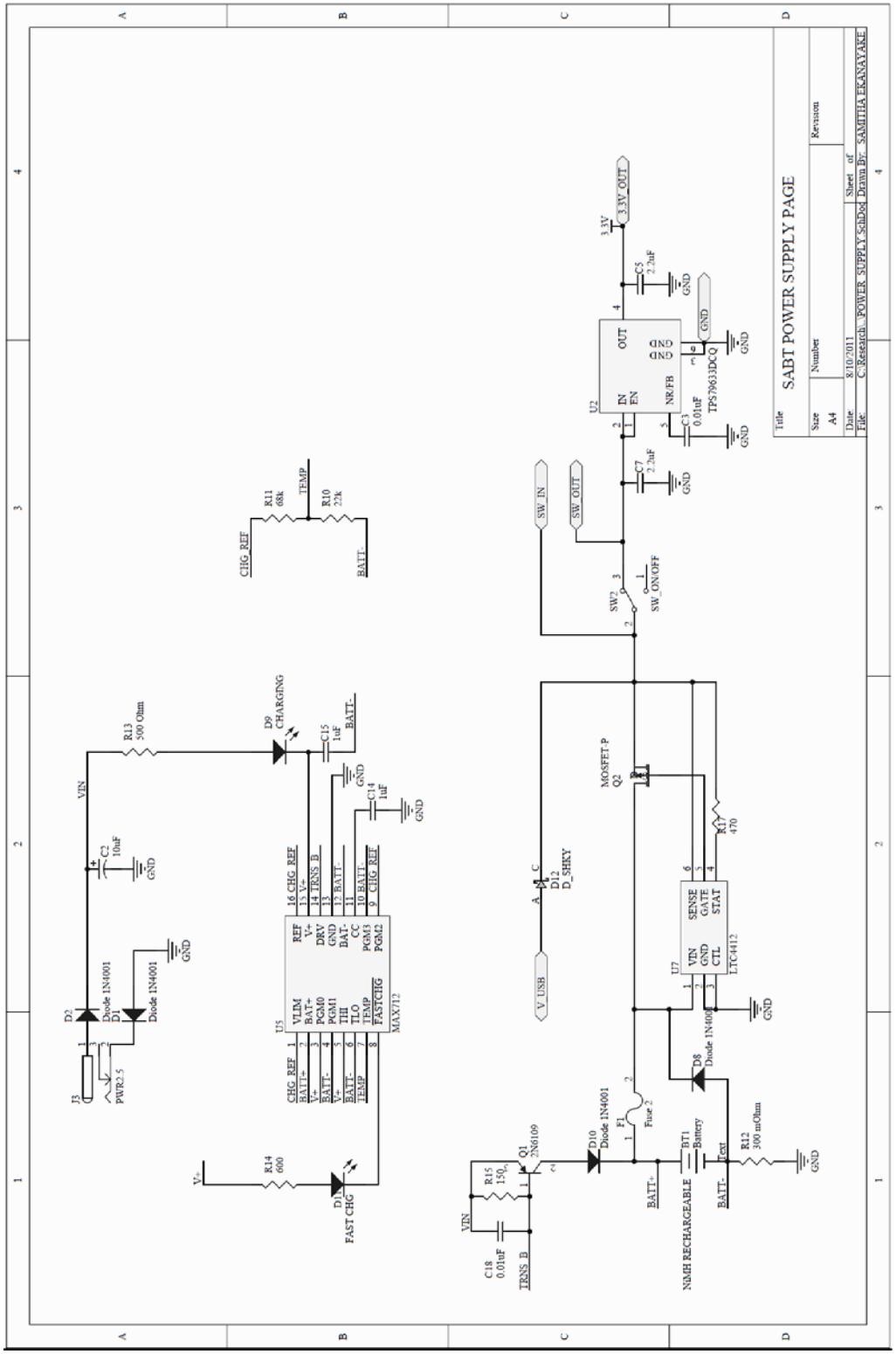


Figure 31: Power management sub-module schematic of the SABT main control module.

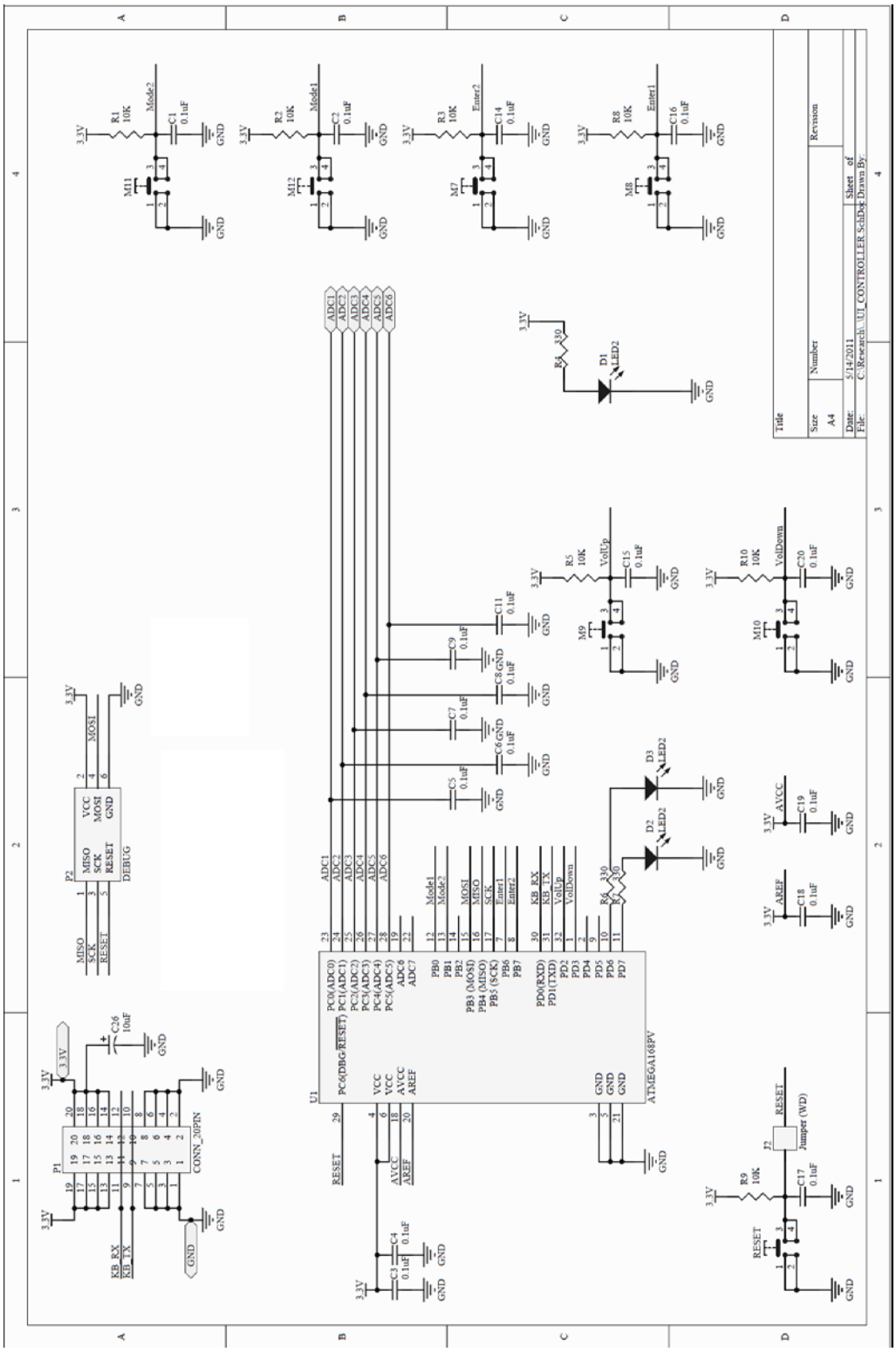


Figure 32: ATmega168PA connections in the SABT user interface module. All the control button interfaces, main controller connector (P1) and ADC input lines from the Braille interfaces are shown here. Note that the ADC lines have a decoupling capacitor in order to prevent line noise generating false Braille inputs.

B Appendix - PCB designs

SABT main control module PCB is a four layer design and the layers are arranged in the following order: TOP ROUTING, GND PLANE, PWR PLANE, BOTTOM ROUTING. Components are placed in both TOP and BOTTOM layers of the board.

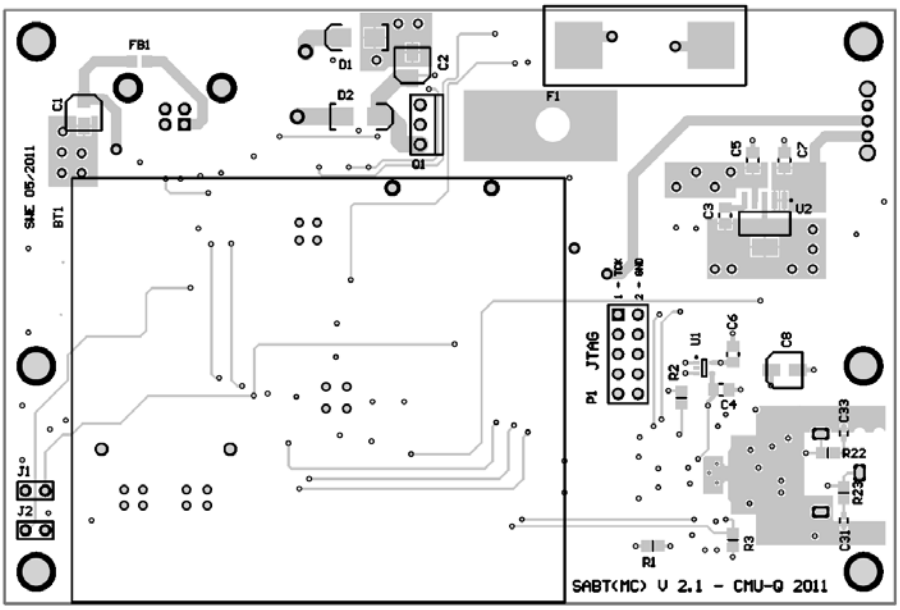


Figure 33: SABT main control module PCB - TOP ROUTING layer.

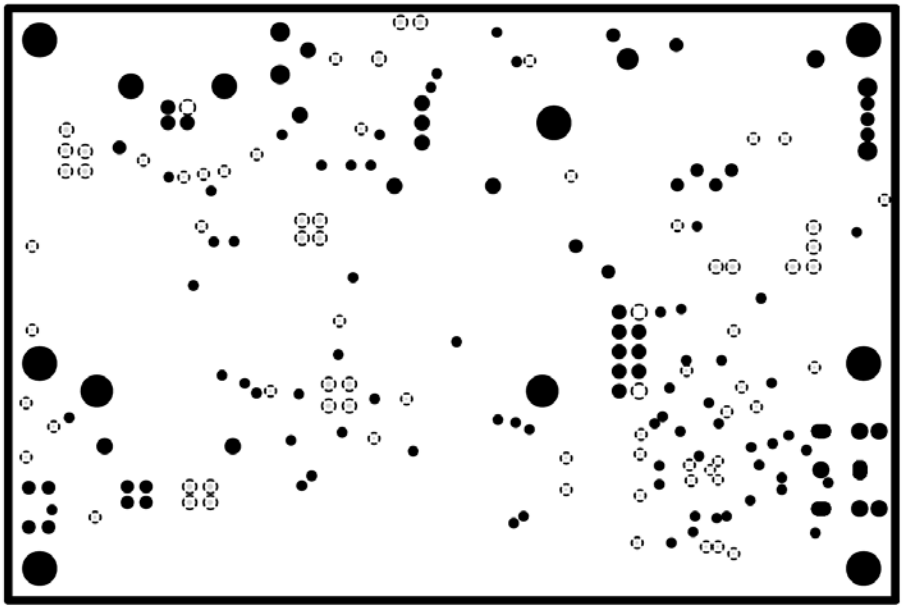


Figure 34: SABT main control module PCB - GND PLANE

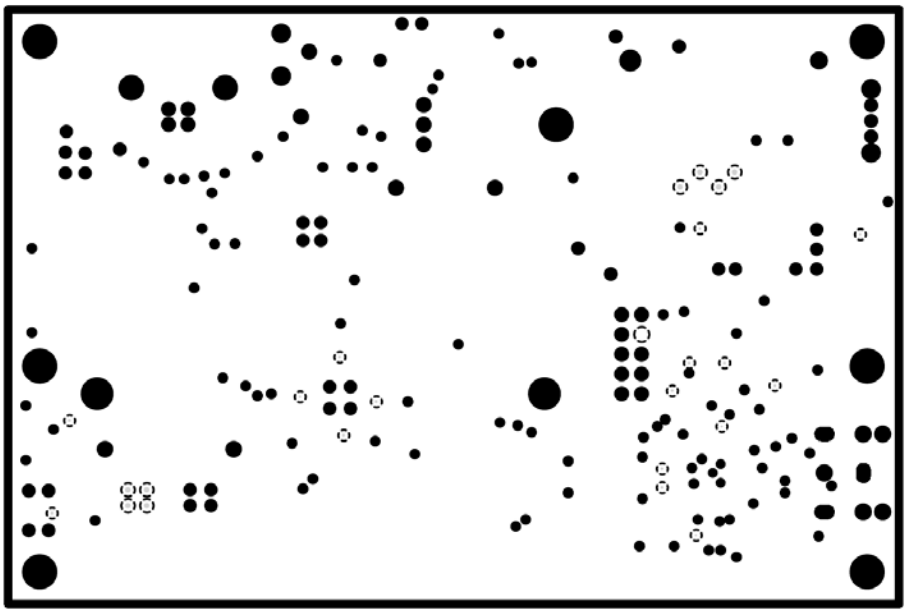


Figure 35: SABT main control module PCB - PWR PLANE

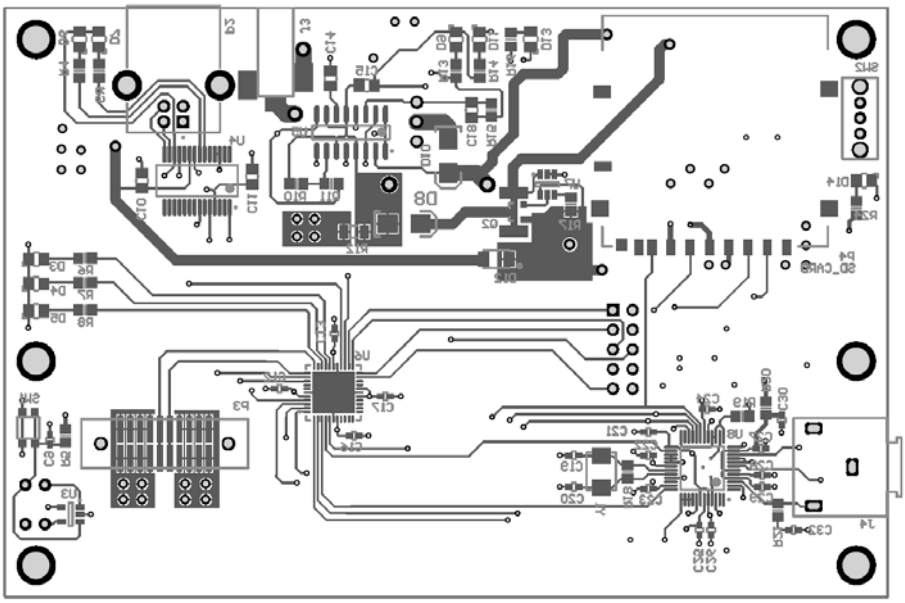


Figure 36: SABT main control module PCB - BOTTOM ROUTING layer.

C Appendix - Modemanager: User's Guide

ModeManager is a software tool designed to compile a set of modes together and create the binary file to be programmed to the SABT main control module. ModeManager is a java program, so your computer need have java runtime environment installed. If you are not certain about this, please visit www.java.com and install the latest version of java in your computer. Extract all the files in the ModeManager.zip file into a single folder. Once you have completed the java installation, run the ModeManger.jar file. Figure **??** shows the startup window of the ModeManager.

Figure 37: Mode manager start screen

A mode comes as a single folder containing all the source and media files in it. Do not modify any of those files inside a mode folder. In order to be identified by the ModeManager, all the mode folders should be placed in a common folder, typically "ModeFiles" folder in the installation path of the SABT software. The path to the folder containing modes is given using the "File¿¿Open mode folder" menu item, which pops a folder open dialog box. Navigate to the appropriate folder and click open.

Figure 38: Use file - open menu item to select the "ModeFile" folder. This will list all available modes in the GUI.

In this example, we have five modes in that folder. The names of the modes will be displayed in the modes box. Pressing "Compile" button in the GUI generates the required source files and copies the other required files to the "src" folder in the "ModeFiles" directory. Also it creates the avr-gcc based compilation commands script file and places it in the "src" folder. In order to compile the SABT source file users need to install avr-gcc library in the system. Then running the compile.bat (in Windows) will create the .hex file to be programed to the device.

D Appendix - ModeSelector: User's Guide

ModeSelector is a software utility designed to select a subset of modes programmed to the device. This software assumes the SABT main controller modes are compiled using the ModeManager software. Also, this assumes that users have not done any modifications to the mode identification files and the folder structure in the "ModeFile" directory. ModeSelector is a java program, so your computer need have java runtime environment installed. If you are not certain about this, please visit www.java.com and install the latest version of java in your computer. Extract all the files in the ModeSelector.zip file into a single folder. Once you have completed the java installation, run the ModeSelector.jar file. Figure **??** shows the startup window of the ModeManager.

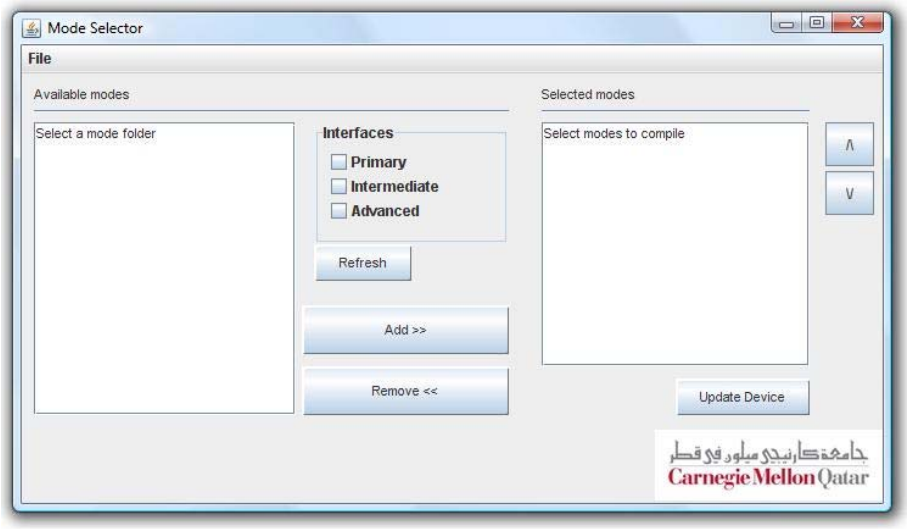


Figure 39: Start screen of the ModeSelector GUI

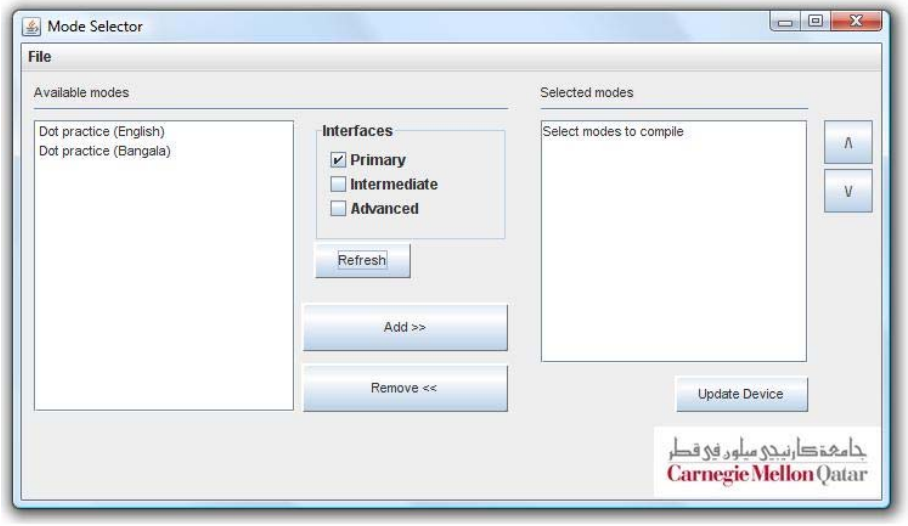


Figure 40: Use "file - open" menu item to open the "ModeFiles" folder. Then select the interface types and press "Refresh" to list the appropriate mode file. Note that modes will not be listed if a user interface type is not selected.

Once the selection and ordering has been completed, press "Update Device" button to update the "modes.dat" file in the SD card of the device. Before moving to this step, make sure the SABT is connected to the computer via the USB cable and the device is switched on. Upon successful completion

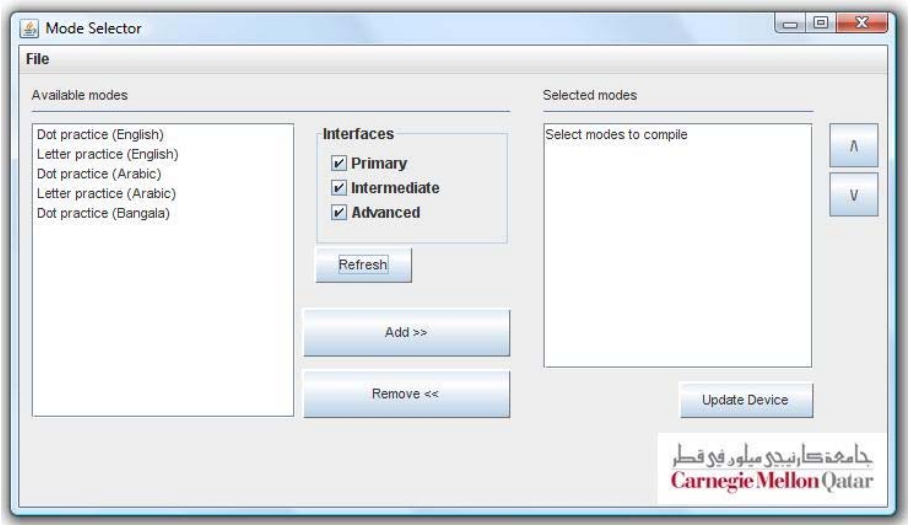


Figure 41: Selecting all the interfaces will list all available modes.

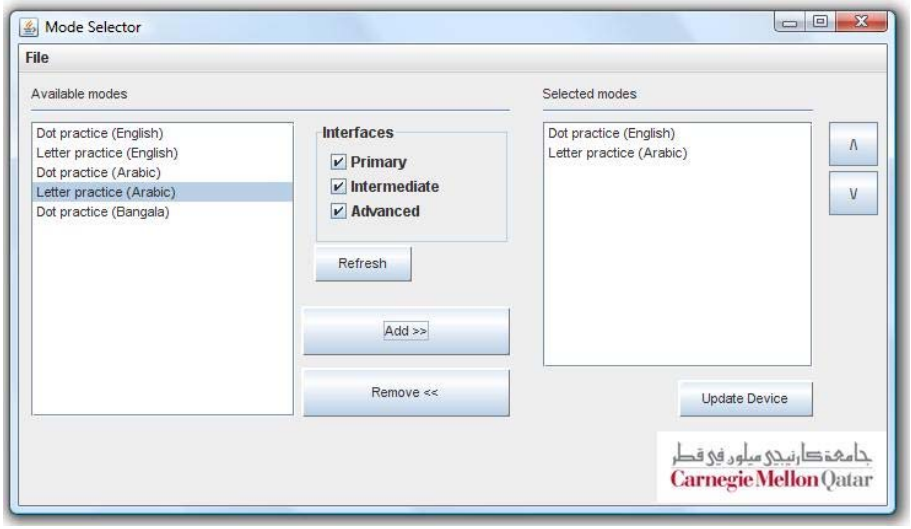


Figure 42: Use "Add". "Remove" buttons to select/de-select modes. Also the order of the selected modes can be changed using the "Up" / "Down" buttons. Simply select the mode that you need to move in the selected modes list and press "Up/Down" button to change the position in the list.

of the process, the "ModeSelector" gives an acknowledgement message asking to power toggle the device to take changes into effect. Disconnect the device from the USB and power toggle.

E Appendix - Code sections

E.1 Structure of a simple mode - “Dot practice”

```
#include "Globals.h"
#include "Modes.h"
/*
Every mode should include the "Globals.h" and the common header file for the modes
*/

int MD$XX$_Current_State;
/*
Current_State - Defines the status of the mode
---Values and meanings---
0 - Just started (in the begining, play the Welcome message)
1 - Waiting for user input
2 - Last user input processed
*/

char MD$XX$_Last_Dot;  //This stores the last pressed dot { used by the playback

/*This functions is called after a dot input was detected, it
Requests appropriate sound file based on the dot pressed.*/
void MD$XX$_PlayRequestedDot(void)
{
  switch(MD$XX$_Last_Dot)
  {
    case '1':
      RequestToPlayMP3file("MD$XX$_1.MP3");
      break;
    case '2':
      RequestToPlayMP3file("MD$XX$_2.MP3");
      break;
    case '3':
      RequestToPlayMP3file("MD$XX$_3.MP3");
      break;
    case '4':
      RequestToPlayMP3file("MD$XX$_4.MP3");
      break;
    case '5':
      RequestToPlayMP3file("MD$XX$_5.MP3");
      break;
    case '6':
      RequestToPlayMP3file("MD$XX$_6.MP3");
      break;
  }
}

void MD$XX$_Reset(void)
{
  MD$XX$_Current_State=0;
}

void MD$XX$_Main(void)
{
  switch(MD$XX$_Current_State)
  {
    case 0:
      RequestToPlayMP3file("MD$XX$_WC.MP3");
      MD$XX$_Current_State=1;
      break;
    case 1:
      break;
    case 2:
      MD$XX$_PlayRequestedDot();
      MD$XX$_Current_State=1;
      break;
  }
}
```

Structure of a simple mode - “Dot practice” (contd..)

```
void MD$XX$_CallModeYesAnswer(void)
{
}

void MD$XX$_CallModeNoAnswer(void)
{
}

void MD$XX$_InputDot(char thisDot)
{
  MD$XX$_Last_Dot=thisDot;
  MD$XX$_Current_State=2;
}

void MD$XX$_InputCell(char thisCell, int iRow, int iCol)
{
}
```

MDXX.h Header file for the mode

```
#ifndef _MD$XX$_H_
#define _MD$XX$_H_

void MD$XX$_Main(void);
void MD$XX$_Reset(void);
void MD$XX$_CallModeYesAnswer(void);
void MD$XX$_CallModeNoAnswer(void);
void MD$XX$_InputDot(char thisDot);
void MD$XX$_InputCell(char thisCell, int iRow, int iCol);

#endif
```

E.2 Source of the UI handle module

UI_Handle.h : Header file the of the UI Handle module

```
#ifndef _UIHANDLE_H_
#define _UIHANDLE_H_

#include <stdint.h>
#include <stdbool.h>

#define UI_CMD_NONE 0
#define UI_CMD_ENT1 1
#define UI_CMD_ENT2 2
#define UI_CMD_MFOR 3
#define UI_CMD_MREV 4
#define UI_CMD_VOLU 5
#define UI_CMD_VOLD 6

volatile bool UI_MP3_file_Pending;
volatile bool UI_MODE_SELECTED;
volatile char UI_Current_Mode;
volatile char UI_Selected_Mode;

//Dealing with the user data
uint16_t UI_calculate_CRC(unsigned char* pstrMsg);
bool UI_parse_message(bool IsPlaying);
void UI_ControlKeyPressed(void);


//Current mode related functions
void UI_Play_Intro_Currentmode(void);
void UI_CallModeYesAnswer(void);
void UI_CallModeNoAnswer(void);
void UI_InputDotToCurrentMode(char thisDot);
void UI_InputCellToCurrentMode(char thisCell);
void UI_RunMainOfCurrentMode(void);
void UI_ResetTheCurrentMode(void);


#endif
```

Function to process received data from User interface

```
#include "Globals.h"
#include "Modes.h"


bool UI_CheckModes(void)
{
  /*
  The modes.dat contains the numbers of modes that need to be activated in this device.
  It contains the mode numbers of the required modes as follows:
  MODENO#MODENO#$ ETC. (Last character is $)
  For example if we need modes 2, 5 and 14 to be activated:
  <2><5><14>$
  */
  unsigned char FileContent[100];
  unsigned char ModeID[3];
  unsigned char PCPrintContent[2];
  int i=0;
  int iMoN;
  bool bBoNFound;
  const char* ModesFile="MODES.DAT";
  Number_of_modes=0;
  for(i=0;i<100;i++)
    FileContent[i]=0;
  if(readAndRetreiveFileContents (ModesFile,FileContent)>0)
  {
  return false;
  }
  USART_transmitStringToPC(&FileContent);
  TX_NEWLINE_PC;
  bBoNFound=false;
  i=0;
  while(FileContent[i]!='$')
  {
    if(FileContent[i]=='>')
    {
      UI_Modes[Number_of_modes]=atoi(ModeID);
      Number_of_modes++;
      bBoNFound=false;
    }
    if(!bBoNFound)
    {
      ModeID[0]=0;
      ModeID[1]=0;
      ModeID[2]=0;
      iMoN=0;
    }
    else
    {
      if(iMoN==3) return false;
      ModeID[iMoN++]=FileContent[i];
    }
  if(FileContent[i]=='<')
    bBoNFound=true;
    i++;
  }
  USART_transmitStringToPCFromFlash(PSTR("Number of modes selected: "));
  PCPrintContent[0]=0;
  PCPrintContent[1]=0;
  sprintf(PCPrintContent, "%d", Number_of_modes);
  USART_transmitStringToPC(&PCPrintContent);
  TX_NEWLINE_PC;
  USART_transmitStringToPCFromFlash(PSTR("And the modes are; "));
  for(i=0;i<Number_of_modes;i++)
  {
    sprintf(PCPrintContent, "%d, ", UI_Modes[i]);
    USART_transmitStringToPC(&PCPrintContent);
  }
  TX_NEWLINE_PC;
  return true;
}
```

Function to process received data from User interface (contd..)

```
bool UI_parse_message(bool IsPlaying)
{
  //include code to interpret the UI string
  /*
  UI string  : [U][I][msglen][msg_number][msgtype][payload][CRC1][CRC2]
  msgtypes :
  */
  //First things first, check the CRC

  unsigned char message_len=USART_UI_ReceivedPacket[2];
  unsigned char message_number;
  unsigned char message_type;

  unsigned char ADCmsg[10];
  //unsigned char message_payload[20];
  // unsigned char i=0;

  uint16_t chksum=UI_calculate_CRC(&USART_UI_ReceivedPacket);
  if ( chksum == (USART_UI_ReceivedPacket[message_len-2] << 8 | USART_UI_ReceivedPacket[message_len-1]))
  {
  //If correct, store the message elements
  message_number=USART_UI_ReceivedPacket[3];
  message_type=USART_UI_ReceivedPacket[4];

  //process the message

  if(IsPlaying && message_type==68) //If a MP3 file is being played, only the commands are processed
  {
    UI_ControlKeyPressed();
    USART_UI_Message_ready=true;
    return true;
  }

  switch(message_type)
  {
    case 65:  //Braille dot
      //Only one character is being send to the current mode
      UI_InputDotToCurrentMode(USART_UI_ReceivedPacket[5]);
      break;
    case 66:  //Braille cell
      /*
      Only one character is being send to the current mode. The cell number value is currently not used
      , if needed this information is available on USART_UI_ReceivedPacket[6]
      */
      UI_InputCellToCurrentMode(USART_UI_ReceivedPacket[5]);
      break;
    case 67: //Error message
      //When an error occured in the user input a message will be sent here
      break;
    case 68: //User Command
      UI_ControlKeyPressed();
      break;
    case 69: //Acknowledgement
      ADCmsg[0]=USART_UI_ReceivedPacket[5];
      ADCmsg[1]=USART_UI_ReceivedPacket[6];
      ADCmsg[2]=USART_UI_ReceivedPacket[7];
      USART_transmitStringToPCFromFlash(PSTR("Analog Input channel,MSB,LSB :"));
      sprintf(ADCmsg, "%d,%d,%d", USART_UI_ReceivedPacket[5],USART_UI_ReceivedPacket[6],
    USART_UI_ReceivedPacket[7]);
      USART_transmitStringToPC(&ADCmsg);
      TX_NEWLINE_PC;
      TX_NEWLINE_PC;
      break;
    default:
      break;
  }
  //In the end: send the acknowledgement to the sender (with the message number, of course !!!)
  }
  else
  {
    USART_UI_Message_ready=false;
    return false;
  }
  USART_UI_Message_ready=false;
  return true;
}
```

Function to process received data from User interface (contd..)

```
void UI_ControlKeyPressed(void)
{
  switch(USART_UI_ReceivedPacket[5])
  {
    case UI_CMD_NONE:
      break;
    case UI_CMD_ENT1: //Enter into a mode
      USART_transmitStringToPCFromFlash(PSTR("Enter 1 pressed"));
      TX_NEWLINE_PC;
      if(!UI_MODE_SELECTED) //Then this command is to select the mode
      {
        if(UI_Selected_Mode>0)
        {
          UI_MODE_SELECTED=true;
          UI_ResetTheCurrentMode();
        }
        else
        {
          //RequestToPlayMP3file("ERR1.MP3");
        }
      }
      else //Then this the "YES" command in the mode, so call the function in the mode
      {
        UI_CallModeYesAnswer();
      }
      break;
    case UI_CMD_ENT2: //Exit from a mode
      USART_transmitStringToPCFromFlash(PSTR("Enter 2 pressed"));
      TX_NEWLINE_PC;
      if(UI_MODE_SELECTED) //This might be an exit from mode command or "NO" command in the mode
      {
        if(USART_UI_ReceivedPacket[6]==69) //If the next byte is 'E', this is exit command
                                           //(when the user pressed E2 for more than 5 secs)
        {
          UI_MODE_SELECTED=false;
          RequestToPlayMP3file("MM.MP3");
        }
        else //Then this a "NO" answer, call the mode function for this
        {
          UI_CallModeNoAnswer();
        }
      }
      //This has no effect when no mode is selected
      break;
    case UI_CMD_MFOR:
      USART_transmitStringToPCFromFlash(PSTR("Mode 1 pressed"));
      TX_NEWLINE_PC;
      if(!UI_MODE_SELECTED)
      {
        UI_Selected_Mode++;
        if(UI_Selected_Mode>Number_of_modes)
        {
          UI_Selected_Mode--;
          UI_Current_Mode=UI_Modes[UI_Selected_Mode-1];
        }
        else
        {
          UI_Current_Mode=UI_Modes[UI_Selected_Mode-1];
          VS1053_SKIP_PLAY=true;
          UI_Play_Intro_Currentmode();
        }
      }
      break;
```

Function to process received data from User interface (contd..)

```
    case UI_CMD_MREV:
      USART_transmitStringToPCFromFlash(PSTR("Mode 2 pressed"));
      TX_NEWLINE_PC;
      if(!UI_MODE_SELECTED)
      {
        UI_Selected_Mode--;
        if(UI_Selected_Mode<1)
        {
          UI_Selected_Mode=1;
          UI_Current_Mode=UI_Modes[UI_Selected_Mode-1];
        }
        else
        {
          UI_Current_Mode=UI_Modes[UI_Selected_Mode-1];
          VS1053_SKIP_PLAY=true;
          UI_Play_Intro_Currentmode();
        }
      }
      break;
    case UI_CMD_VOLU:
      USART_transmitStringToPCFromFlash(PSTR("Vol UP pressed"));
      TX_NEWLINE_PC;
      VS1053_IncreaseVol();
      break;
    case UI_CMD_VOLD:
    USART_transmitStringToPCFromFlash(PSTR("Vol DOWN pressed"));
      TX_NEWLINE_PC;
      VS1053_DecreaseVol();
      break;
    default:
      break;
  }
}

uint16_t UI_calculate_CRC(unsigned char* pstrMsg)
{
  unsigned char msglen=*(pstrMsg+2)-5;//Not including the checksum bytes
  uint16_t chksum=0;
  pstrMsg+=3;
  while(msglen > 1)
  {
    chksum+=(*(pstrMsg)<<8) | *(pstrMsg+1);
    chksum = chksum & 0xffff;
    msglen-=2;
    pstrMsg+=2;
  }
  if(msglen>0) //If the packet size is odd numbered
    chksum = chksum^ (int)*(pstrMsg++);
  return(chksum);
}

bool UI_buildMessage(char MessageType)
{

return true;
}
```

Function to process received data from User interface (contd..)

```
void UI_Play_Intro_Currentmode(void)
{
  switch(UI_Current_Mode)
  {
    case 1:
      RequestToPlayMP3file("MD1.MP3");
      break;
    case 2:
      RequestToPlayMP3file("MD2.MP3");
      break;
    case 3:
      RequestToPlayMP3file("MD3.MP3");
      break;
    default:
      break;
  }
}

void UI_CallModeYesAnswer(void)
{
  switch(UI_Current_Mode)
  {
    case 1:
      MD1_CallModeYesAnswer();
      break;
    case 2:
      MD2_CallModeYesAnswer();
      break;
    case 3:
      MD3_CallModeYesAnswer();
      break;
    default:
      break;
  }
}

void UI_CallModeNoAnswer(void)
{
  switch(UI_Current_Mode)
  {
    case 1:
      MD1_CallModeNoAnswer();
      break;
    case 2:
      MD2_CallModeNoAnswer();
      break;
    case 3:
      MD3_CallModeNoAnswer();
      break;
    default:
      break;
  }
}

void UI_InputDotToCurrentMode(char thisDot)
{
  switch(UI_Current_Mode)
  {
    case 1:
      MD1_InputDot(thisDot);
      break;
    case 2:
      MD2_InputDot(thisDot);
      break;
    case 3:
      MD3_InputDot(thisDot);
      break;
    default:
      break;
  }
}
```

Function to process received data from User interface (contd..)

```
void UI_InputCellToCurrentMode(char thisCell)
{
  switch(UI_Current_Mode)
  {
    case 1:
      MD1_InputCell(thisCell);
      break;
    case 2:
      MD2_InputCell(thisCell);
      break;
    case 3:
      MD3_InputCell(thisCell);
      break;
    default:
      break;
  }
}

void UI_RunMainOfCurrentMode(void)
{
  if(UI_MODE_SELECTED){
    switch(UI_Current_Mode)
    {
    case 1:
      MD1_Main();
      break;
    case 2:
      MD2_Main();
      break;
    case 3:
      MD3_Main();
      break;
    default:
      break;
    }
  }
}

void UI_ResetTheCurrentMode(void)
{
  if(UI_MODE_SELECTED){
    switch(UI_Current_Mode)
    {
      case 1:
        MD1_Reset();
        break;
      case 2:
        MD2_Reset();
        break;
      case 3:
        MD3_Reset();
        break;
      default:
        break;
    }
  }
}
```