Table of Contents

1.0 Introduction

Welcome to the PETI Project! If you are reading this, it is most likely that you have acquired (or intend to produce) a PETI Development Kit. PETI – the **P**et-like **E**mbedded **T**echnology **I**nstructor – is a long running project from us at Arcana Labs to create a virtual pet that serves as a hardware and firmware demonstrator for embedded design principles. From the outset, it's intended to be hackable, expandable, and entirely open down to its bare bones.

1.1 Acknowledgements

We at Arcana Labs would like to thank everyone whose ever shown an interest, put in work, or shared opinions and information related to PETI's development cycle. This is truly a community-effort project, and the heritage of that project should very well point back toward our community.

In particular, we would like to thank the following individuals for directly sponsoring PETI through financial contributions:

- Adam Meek (Github: adamwmeek)

Additionally, the following individuals have all made direct contributions in lines of code to the main PETI firmware repository, either via PR or through active discussions:

- Zac Adam-MacEwen//Patch Arcana (Github: zadammac)
- Gen (Github: kosmogen)
- Stephanie Gawroriski (Github: XerTheSquirrel)

Finally, we'd also like to thank the entire Arcana Labs Discord community (in particular the various members of Sponsor's Corner) for their past and ongoing support of this and other Arcana Labs projects!

1.2 Prior Art

In a sense PETI is nearly nothing but prior art, apart (of course) from the parts that were reinvented. The look, feel, and behaviour of the device is inspired in large part by the virtual pets of the latter half of the 1990s, in particularly the Bandai Tamagotchi.

In a much more real sense (especially pertinent in the case of the Development Kit), much of the development kit is made possible by the usage of an existing development kit for the MSP430FR5994 Microcontroller, produced by Texas Instruments. Texas Instruments also produces an accessory board for that same "LaunchPad" development kit, which is used in our kit as a means to interface the LCD screen.

Finally, while the game-play and operations logic are wholly created by Arcana Labs, this was greatly facilitated by the inclusion of headers and driver libraries supplied by Texas Instruments to streamline the development process using their microcontrollers.

1.3 Project Goals

There are essentially three goals for the PETI development project:

1. Produce a single-board object with a similar gameplay experience to the virtual pets of the 1990s;
2. In a manner as consistent as possible with the principles of OSHW and FOSS, to the extent that the final device should be hackable, expansible, and reproducible, and;
3. This being Arcana Labs, hide a few puzzles along the way.[1]

1.4 External Resources

In addition to this guide, we recommend making use of the following resources:

- Texas Instruments' Website supplies copies of their driver files along with your choice of two Integrated Development Environments (IDEs) configured from installation to work nicely with the LaunchPad at the heart of your development kit. Of the two, we recommend **Code Composer Studio,** which contains files needed for the makefile to do its job when using your actual IDE of choice.
- TI also publishes many volumes of documentation on the MSP430FR5994, MSP430 Driver Library, and general programming guidelines for their devices. It is recommended to obtain these documents for reference when working with the development kit as they will go into greater detail regarding the very specific behaviour of hardware registers than will be covered here.
- The Arcana Labs Website (https://www.arcanalabs.ca) contains full build logs for the entirety of the PETI project, pursuit of which could lead you to avoiding some of the same pitfalls we originally ran into.
- An additional repo of tools developed explicitly for this project is maintained at https://github.com/zadammac/peti-helpers; especially, a forked version of font-edit which is used to generate fonts for the device, and a dedicated interpreter for the MLCD SPI data. Apart from being called out as the need to reference them specifically arises, those tools are only quasi-supported and are not documented in this Service Manual. Please consult that repo for additional information.

1 At the time of writing of this version of the document, there are three such undocumented features planned.

2.0 Hardware Documentation

2.1 Assembled Device Overview

2.1.1 Controls

The core launchpad board (red) contains three momentary switches (“buttons”) which have a fixed value that are interpreted globally.

Table 2.1.1-A: Launchpad Buttons

Silkscreen Label	**Usage**
RST S3	Device Reset (BOR)
P5.6 S1	Unused (Available for Diagnostics)
S2 P5.5	Unused (Available for Diagnostics)

The control daughterboard represents a special case by comparison to other inputs on the device; these are handled at the Scene Level (See Section “Understanding the Scenes Model” in the Programming Reference portion of this guide.) Because of this they have no “universal” use. It’s considered advisable to stick as close to this pattern as possible or otherwise be clear in the UI what each button is expected to do. The Rock Paper Scissors minigame is a good demonstration of this principle.

Table 2.1.1-B: Control Daughterboard Buttons

Position	**Usage-in-General**
Top Left	Scroll Up/Cycle Forward/Increment (Button “A”)
Bottom Left	Scroll Down/Cycle Backward/Decrement (Button “B”)
Top Right	Accept/Continue (Button “C”)
Bottom Left	Reject/Cancel (Button “D”)

The Expansion Daughterboard (as of Revision C) contains two sets of 2x1 DIP switches to act as interrupts and configuration flags for the device. In general these should be set prior to a reset; however the Hard Mute switch can be used at any time.

Table 2.1.1-C: Expansion Board Switches on Revision C

Switch Label	**Usage**
SW1 A1	Power (Cuts connection between battery holder and the 3.3v Rail)
SW1 A2	Not Connected
SW2 A1	Hardware Mute (Speaker is only usable when switch is in the ON position)
SW2 A2	Debug Mode Enable (When switch is in the ON position)

In Revision D of the Rear Expansion Board replaces this pair of switches with a single switch package (SW1) containing three switches. Presuming that the expansion header is oriented toward the top of the board, those switches operate as follows:

Table 2.1.1-D: Expansion Board Switches on Revision D

Switch Position	**Usage**
Top	Power (Cuts connection between battery holder and the 3.3v Rail)
Middle	Debug Mode Enable (When switch is in the ON position)
Bottom	Hardware Mute (Speaker is only usable when switch is in the ON position)

2.1.2 Assembly Instructions

Ultimately assembly is up to the user's discretion. However, if working with the public schematics for the boards in question or the officially-produced Development Kit by Arcana Labs, the following recommendations are made:

- If using the Rear Expansion Board, it is recommended to use standoffs to secure the corners of the LaunchPad and the Rear Expansion Board together, though not strictly required
- Always assemble the Control Daughterboard (if being used) to the BOOSTXL-SHARP128 LCD Board before installing to the launchpad; this reduces problems with bent header pins.
- If direct interrogation of the pins is desired (e.g. in the case of connecting a logic analyzer) it is recommended to omit the control daughterboard and attach directly to the pins you are interest in from the front of the LCD module.
    - If you are interrogating the LCD module, an LCD_CS pin is provided on the daughterboard (revision C and later) which mirrors the actual LCD chip select signal used in the main device bus. This pin header is marled LCD_CS and is located on the top edge near the EXP_SPI header for convenience. It will not appear in the production model device.

2.1.3 Jumper Configuration for Various Modes

The following Jumpers are all present on the Launchpad (Central Red Board). In general, their silkscreen labelling will be used as shorthand throughout the guide to describe their function. In cases where the jumper label appears duplicated, refer to the dividing line between ez-FET Rev 1.3 and MSP430FR5994 on the board – they make up a header along that line. The table on the following page discusses their usage.

Table 2.1.3-A: Jumpers

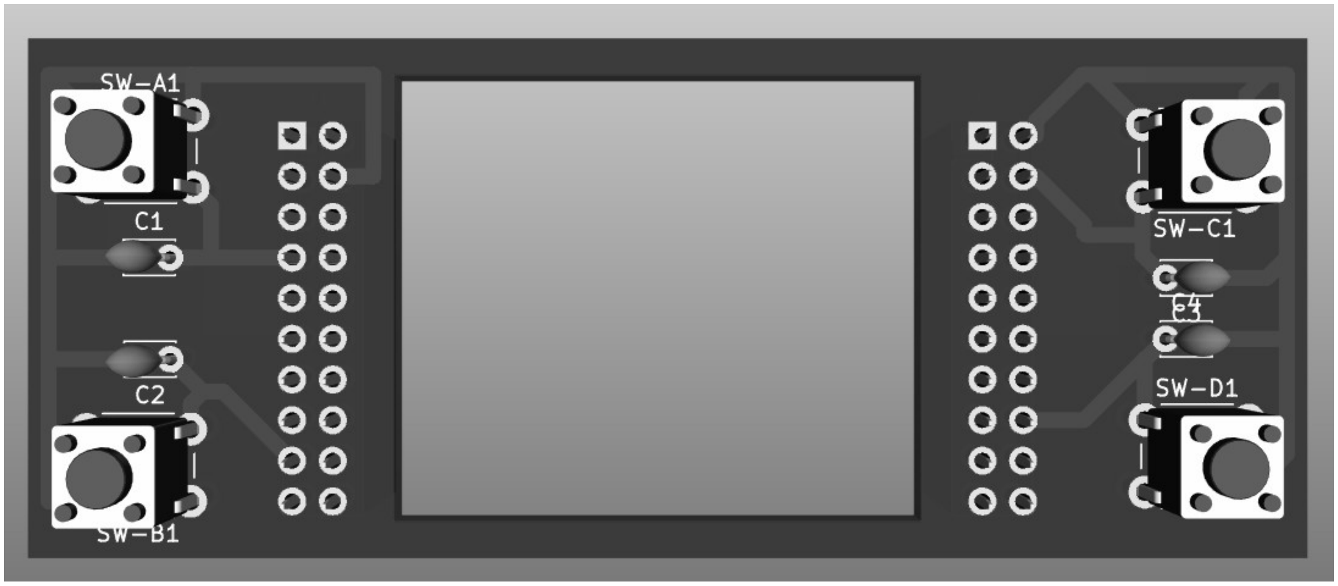
Silkscreen Label	**Usage**
GND	Completes Ground Rail Circuit to USB.
5V	Completes 5v Rail Circuit to USB.
3V3	Completes the 3.3v Rail Circuit to USB
RXD <<	Data Exchange Line Used In Device Programming and Console Usage
TXD >>	Data Exchange Line Used In Device Programming and Console Usage
SBWTDIO	Data Exchange Line Used In Device Programming and Console Usage
SBWTCK	Data Exchange Line Used In Device Programming and Console Usage
J7	Breaks the circuit to the P1.0 Diagnostic LED (Default: Indicates Device Heartbeat and VCOM state.)
J8	Determines if the on-board super capacitor is charging or not. Not used.

During Standalone Mode, when the development kit is being used on battery power, it is required to first remove the 3v3 jumper (placing it over only one of the pins so it does not get lost). This is required to keep the powered 3v3 rail from powering up portions of the eZ-FET board which would cause unnecessary battery train. Refer to TI's Documentation, MSP430FR5994 LaunchPad Development Kit (MSP-EXP430FR5994) User's Guide (Rev. B)**.** Some users may prefer to remove all seven pins in this header and move them over to one side or the other as a sanity check; at minimum the 3v3 and GND pins should be un-jumpered for the device to even fully power on under battery power.

During Tethered Mode, when the development kit is being programmed, it is required that all seven jumpers in the development header (the first seven pins listed in the table above) be in their closed position (i.e. covering both pins). This allows the USB portion of the board to interact with the microcontroller appropriately. However, **because this causes the 3.3v rail to become energized**, it is strongly recommended to either open SW1A1 to cut the circuit to the batteries *or* to remove them outright as a precaution against over-voltage conditions, which may be dangerous depending on your battery chemistry.

2.2 Control Daughterboard

2.2.1 Illustrated Reference (Revision B)



The above image is a 3D rendering of the Revision B version of the daughterboard, albeit one in which the silkscreen is fully rendered. The markings on the buttons (SW-B1, SW-C1, etc) indicate which programmatic “button” they are mapped to in the Pet code. A central hole permits the LCD on the BOOSTXL-SHARP128’s display to be viewed through the board as it would otherwise be blocked.

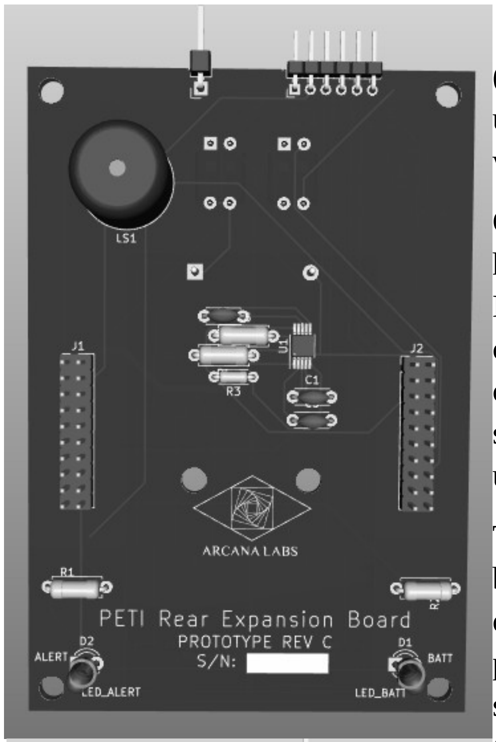
Depending on the production run these markings may not be available. A small number of early-manufacture Revision A control boards did not include the full silk-screening work.

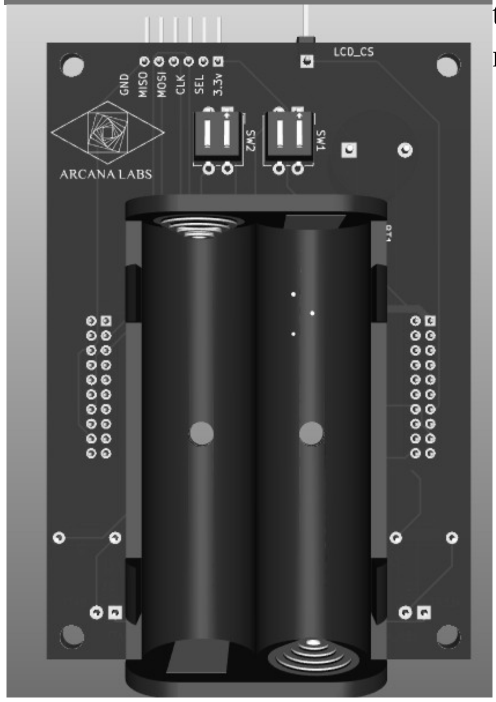
2.2.2 Schematics

The following page is a US Letter rendering of the schematic for the controller board.

2.3 Rear Expansion Board

2.3.1 Illustrated Reference (Revision C)



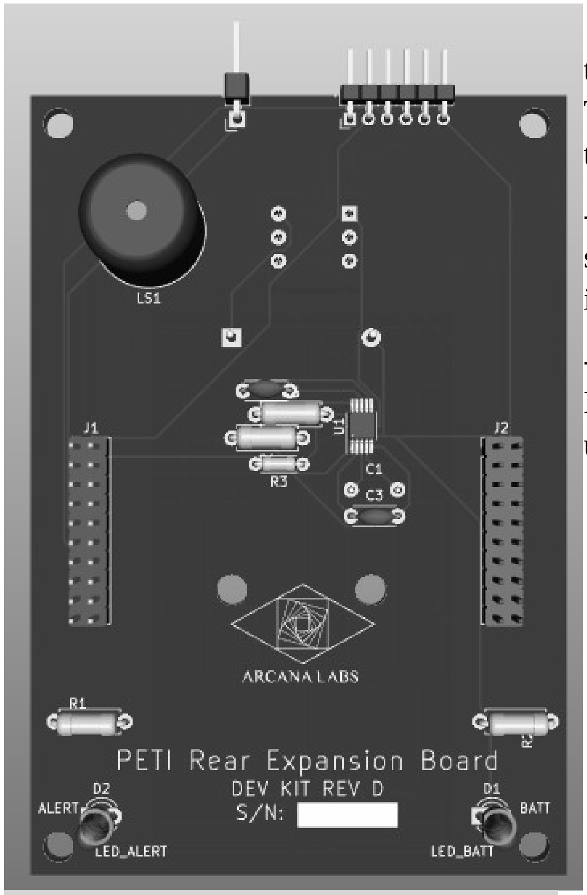


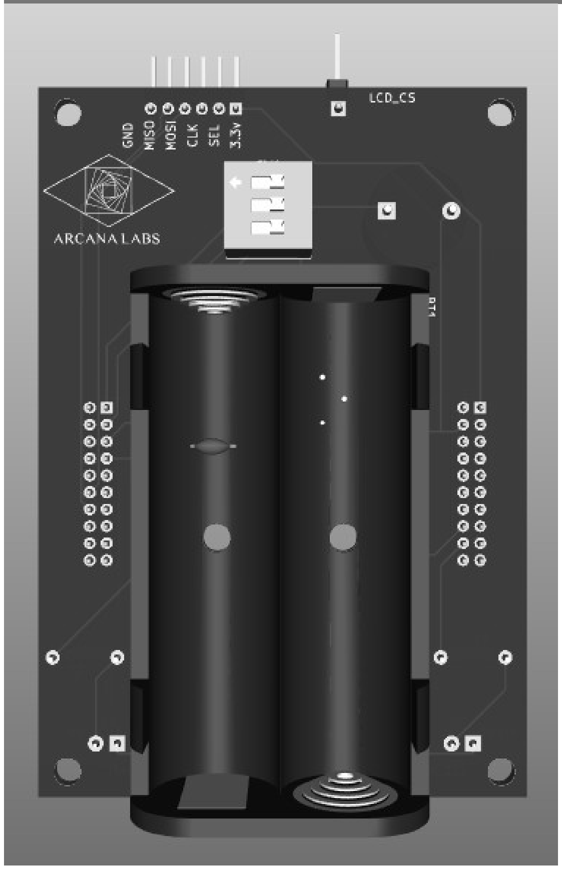
The image to the left is the "front" of the Rear Expansion Board (Rev. C). This is the board that was currently in production and use at the time of the development of Firmware Version 0.2.0, which is the last time this section of the manual was updated.

Of particular note on this side of the board is presence of the two headers J1 and J2 which are used to attach the device to the Launchpad, and the two alert LEDs D2 and D1 in the bottom corners. Holes have been drilled in the corners to provide a convenient means to attach standoffs for further structural support, though this step may well be unnecessary for typical uses.

The image which follows is the rear of the same revision of the board, showing prominently the pinout for Expansion Bus (top of device) and LCD Chip Select pin. It also shows the switch positions. Note that while SW2 and SW1 are labelled in silkscreen, your individual switch packages may not number them. Oriented as they are in the image, the switches should be numbered left-to-right.

2.3.2 Illustrated Reference (Revision D)





The two images on this page show the Revision D design of the board, which will ultimately supplant the revision C. There are only two noteworthy points of difference compared to revision C:

- The replacement of the switches on the back side with a single three-gang switch, the operation of which is described in **2.1.1 Controls,** and;

- The re-arranging of the contact pads on the LEDs D1 and D2 to align with the convention that the square pad belongs under the positive lead of the diode.

2.3.3 Schematics

The following two pages are US-letter printouts of both revisions of the development kit schematic.

2.4 Texas Instruments Hardware In Use

In addition to the two boards supplied as part of your development kit as-purchased (if you originally purchased it from Arcana Labs) or which you had milled from our supplied production files (if you produced the hardware yourself), the development kit relies on two pieces of development hardware produced by Texas Instruments and sold under a not-for-redistribution license. Because TI will not grant Arcana Labs a license to redistribute these modules, they cannot be included in your kit and will have to be obtained separately.

2.4.1 Parts Numbers

The two Texas Instruments components are:

- MSP-EXP430FR5994 “Launchpad” development kit for MSP430FR5994;
- BOOSTXL-SHARP128 development “booster pack” for Sharp mLCDs.

3.0 Programming Reference

This portion of the Service Manual will focus on a high-level overview of the activity of programming using the PETI Development Kit. Following a discussion of development environment considerations, attention is being paid in particular to understanding the logic and workflow of the current version of the firmware.

This Programming Reference is valid as of firmware version 0.2.0, but as always, consult the Firmware Printout section of the guide for more granular information, or refer to the main PETI repo at https://github.com/zadammac/PETI for the latest documentation and firmware source code.

3.1 Recommended Development Environment

Ultimately, as is the general truth of computing, you will be able to configure your environment more or less however you like. However, we officially recommend the use of Code Composer Studio as supplied by Texas Instruments, as it is fully cross-platform and comes pre-configured for use with the MSP430. Configuring your IDE of choice to work directly with the PETI Development Kit is currently beyond the scope of this guide. If strongly desired, the recommendation would be to consult TI’s documentation for the MSP430-EXP5994 Launchpad to determine the appropriate additional settings required to compile to their assembly language and flash the resulting code to the device.

Regardless of how else it is achieved, prospective developers must obtain both the MSP430 Development Headers and the MSP430 Driver Library from Texas Instruments as the source code currently has a dependency on those objects. If you are not using Code Composer Studio or Energia, you will also need to obtain the compilation, linking, and debug tools. Consult the file `documentation/env_setup.md` from the main repo for guidance on setting this up.

3.2 Configuring the Development Kit For Programming

Once you have assembled a development kit and installed the daughter boards to your Launchpad, you will need to flash the device. This is equally true if you have been using a development kit for some time and want to install an updated firmware or are working on a firmware revision of your own:

1. Either open SW1A1 on the Rear Expansion Board to disconnect the battery pack or remove the batteries from the holder entirely.
2. Ensure that all the jumper connections discussed in **2.1.3: Jumper Configuration For Various Modes** as part of the development header are in their closed position (covering both pins).
3. Open a terminal session (in your IDE or through a terminal emulator) at the root of the PETI repository.
4. Run the command `make target-flash`. This will both compile the source code you currently have to an executable binary and flash that binary to your PETI device.

For more information about interacting with the MSP430 via eZ-FET for programming, consult the **MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide** as published by Texas Instruments which details these considerations in depth.

Note: It should be possible to jumper the "top" pins on the development header to the corresponding pins on, say, a standalone PETI board to use the eZ-FET as a programmer. This is not officially supported by Arcana Labs at the time of writing as it has not been fully tested, as the standalone PETI device has not yet been designed.

3.3 FOSS Firmware Contribution Guide

The Firmware for PETI is being provided as Free, Open Source Firmware under the terms of the GNU General Public License (GPL) v 3.0; some sub-elements of the firmware (specifically those provided by Texas Instruments, such as the contents of `src/driverlib`) may have their own license terms. Under the terms of this license, it is up to you whether or not you want to make your modifications public. However, if you do, the GPL does bind you to continuing to include the GPL notice in your code and to make clear your changes.

This section is not going to be a full re-explanation of the GPL, which was included earlier in the document. The subsections of this section will also presume that you are working on PETI firmware with the intent to submit your modifications to the greater community, and are there for your ease of understanding the layout of the firmware repository. However, we also want to encourage you to throw the whole thing out. The PETI hardware is itself a complete and valid circuit (or set of circuits) which could be made into a viable jumping off point for a variety of other projects, even without the underlying firmware.

Regardless of if you are making private modifications for your own amusement, working on the PETI project as a contributor, or building something entirely different out of our leftover parts, we'd love to hear about it! Send an email to peti-derivatives@arcanalabs.ca with the details, or open a PR at https://github.com/zadammac/peti.

3.3.1 Code Formatting and Naming Conventions

At the time of writing, no firm style guide has yet been published for PETI contributions. As a general rule, outside contribution to the project had been limited up until around the release of version 0.2.0 of the firmware. That being said, here are some general guides:

- Wherever possible, line lengths are being kept to a reasonable width (~120 characters or so) for ease of reading.
- Wherever possible, functions belonging to a specific submodule are being given names beginning with an all-capitals designation that makes that association clear, ex. "SCENES_" for the scene manager.
    - For functions that will be exposed in the corresponding header file and referenced in other modules, this is effectively mandatory after v.0.0.4; some legacy code may not yet obey this.

- Wherever possible, global objects are being named in all capital letters, for example "DISPLAY_FRAME" for the global DisplayFrame struct instance.
    - As of versions 0.0.8 and later, this is effectively mandatory for new contributions, and exceptions will be refactored away prior to version 1.0's release.
- Wherever possible, avoid 'magic numbers'[1]. The display.c driver file has a good example of avoiding this by use of nested defines. The more that can be derived from the fewer magic numbers, the easier ports of the firmware to other MSP430-family microcontrollers or other display configurations will be.

3.3.2 Project Structure Overview

If you have recently cloned the PETI repo (and have correctly pulled in the driver library and msp430 headers) you will notice it has a very specific layout:

```
PETI/
├─ cad_sketches/
├─ documentation/
├─ hardware_design/
├─ src/
│  ├─ main.c
│  ├─ main.h
│  ├─ driverlib/
│  ├─ lib/
├─ targetConfigs/
├─ Makefile
├─ localsettings.mk
```

Of these folders the following can all be said:

- **documentation** houses (ideally) current documentation for all functionality. Prior to the release of v.0.1.0 this is likely to be significantly outdated.
- **src/driverlib** is the source code for TI's MSP430 Driver Library. It's included in the repo only as a convenient way to freeze a specific moment in time. We recommend using this version over whatever the latest public version is as updates to this package have been known to introduce breaking changes.
- **hardware_design** contains notes, schematics, pcb layout, and in some cases gerber files for any publicly-released hardware as part of the project, including the hardware for the PETI Development Kit.
- **src/lib** contains the bulk of the game's source code, laid out as follows:
    - **alerts,** which contains convenience functions for handling audio and LED alerts as well as the more primitive functions that are used to build those functions;

1 Magic Numbers are directly hardcoded values (usually, but not always, integers) which are included inline to other code to make them work. Wherever possible these should be replaced with a `#define` statement near the top of the relevant document instead.

- **display,** which contains the source code for display-related objects including the primary display drivers (display.c), graphical fonts, scene layout definitions, and font source files. Refer to a later section of this guide, **Understanding Display State Management**, for more details.
- **game**, which contains various game-state tracking code and objects such as the game manager, evolutionary data, and food data. See the sections later in this guide, **Understanding Game State Initialization, Understanding the PETI PRNG** and **Understanding Game State Updates** for more information.
- **hwinit** contains the core logic for initializing the hardware in the development kit to do it's job, such as by configuring GPIO pins appropriately. See **Understanding the Hardware Init Process** and **Handling Human Input** for more details.
- **locales** contains a central set of locale-specific header and source files so that all necessary strings (as would be seen by a player) can be abstracted away in the source code, allowing for better portability between human-read languages. See **Understanding Locale Files** later in this guide for more details.
- **Menus** contain menus as implemented for either of the Menu Generators (see **Understanding the Menu Generators**). Implementing menus in this way allows for greater code reuse than implementing them as bespoke scenes, and breaking them out into their own directory makes it clearer which menus are being serviced by the generators and which are still bespoke for future code improvements.
- **Scenes**, which contains sub-directories as necessary for assets used by individual scenes in the game. As scenes are a highly central mechanic to how the device operates with its official firmware, they are discussed in much greater detail in the section **Understanding the Scenes Model**.

3.4 Understanding PETI Operational Concepts

3.4.1 The Main Program Loop

Ideally, for most of the wall-clock time, the MSP430 microcontroller in PETI is actually sitting idle in one of its low power modes. When an interrupt is detected – either from TIMER_A0 or from a human input event – it is put back into normal running mode and begins at the top of the "main gameplay loop", or main program loop, which is a `while true` loop in the function `main` of `main.c`. This gameplay loop does the following things, in order:

1. Disable LPM X.5 for arcane hardware reasons[1];
2. Call a function from the Game Manager to evaluate any outstanding timed events.
3. Check if the interrupt was from human input, and if so, clear the alert LED.
4. Call the Scene Manager (see, among others, **3.5.4 Understanding the Scenes Model**) to draw the next 1 frame of screen animation.
5. Toggle the VCOM state, which "flips" polarities within the LCD module to prevent stuck pixels.
6. Force a shuffle of the PRNG to prevent memorization of the random state.
7. Set the flag used for step 3 back to `0x00`[2]
8. Go back into Low Power Mode

All being well this loop keeps the game state and display ticking over while allowing the MSP430 to mostly sit in its lowest-power mode, which is important for planned future work around improving power management on the device.

3.4.2 Understanding the Hardware Init Process

Prior to beginning primary operation – indeed, well before a human observer could likely notice the delay – several events take place in order to place the device in a working state, namely:

1. Disabling the Watchdog Timer, which is currently unused. Failure to do so will result in an unhandled interrupt which would lock the device up until it is manually reset.
2. Disabling LPM X.5, which in some circumstances would cause pins to get stuck in either the high or low state.
3. Initializing some device state flags (VCOM, FORCE_REFRESH, and interacted_flag).
4. Initializing all used GPIO pins such that the appropriate pins are either being used as pulled-up input resistors or are preconfigured to be used for outputs.

1 This is precautionary and may not be required.
2 The language that the firmware is written in, C, does not have a native boolean type. Integers are instead used, with the convention of zero being false.

5. Initializing all used timer registers so that they are available to use, including setting the system clock to its current default of 8 MHz and setting up Timer A0CCR0 (the heartbeat timer) to interrupt roughly twice per second.
6. Initializing the RTC by setting the time to a default value defined in the uppermost lines of hwinit.c, and letting the RTC run.
7. Initializing SPI by establishing the eUSCI B1 peripheral as a SPI Main nominally pegged to 1 MHz, which is the theoretical maximum of the mLCD panel. This SPI bus is to be shared by the eventual expansion bus.
8. Finally, initialize the LCD panel by beginning the regular heartbeat of VCOM states and sending the screen clear signal (if this is not done quickly enough, random garbage patterns are printed to the screen instead).

Once these steps have completed and the game state is initialized, the main program loop (see section 3.4.1) takes over.

3.4.3 Understanding Game State Initialization

As a final step before handing operation over to the main game loop, main.c calls GAME_initStateStruct from game_manager.c, setting some initial state members in the global StateMachine structure:

- Sets the pet's age, current hunger, boredom, and discipline, hidden naughtiness statistic, stage ID, and health byte to 0. This accurately describes the starting state of an egg and its freshly-hatched baby PETI version.
- Sets a global integer flag, egg_delay_set, to 0, indicating that the correct transition offset for the egg hatching had not yet been set.
- Sets an initial integer egg_delay to 1, causing later functions to apply a one-minute transition delay between that moment and the egg hatching.
- Initializes the global NEXT_STAGE_TRANSITION values to 0 as a sanity check, and, finally;
- Sets the global indicator SCENE_ACT to the defined Scene Address for the boot display.

As this function returns its void, we enter the main game loop.

3.4.4 Understanding Game State Updates

The Game Manager also provides a convenience function, GAME_evaluateTimedEvents, that is called every loop of the main game loop prior to handling inputs and rendering the screen. Within this function:

1. The RTC_C Peripheral is interrogated for the current in game system time.
2. If the calendar_initial_setup_completed flag is set to true but the egg_delay_set flag is not true, the NEXT_STAGE_TRANSITION_MINUTES and NEXT_STAGE_TRANSITION_HOURS

values are updated according to the egg_delay int defined elsewhere in the file, and the egg_delay_set flag is set to true.

3. If all of the current NEXT_STAGE_TRANSITION values work out less than their “current” equivalents, and the egg_delay_set flag is true:
    1. An evolution function is called to determine the next stage of evolution to use; this is not actually implemented as of v 0.2.0.
    2. The screen is forced to refresh by setting the FORCE_REFRESH flag. This was required by the “older” display system and may be removed in a future release.
    3. Temporarily, NEXT_STAGE_TRANSITION_AGE is set to 0xFF to avoid looping as evolution is not currently fully implemented.
4. If the current seconds value is 0, and we have reached the top of the minute, the various game state manipulation functions (such as hunger and fund degredation, application of an illness or death state, etc) are applied.

3.4.5 Understanding the Scenes Model

In order to simplify the main gameplay loop, PETI relies on a system of **scenes**, which correspond to a specific set of display states. Put simply, a “scene” is anything that would be displayed on the screen. The main gameplay screen is a scene. The calendar menu is a scene. The food menu is a scene, etc.

This is achieved in the following way:

- scene_manager.h contains a series of defines which give a unique integer value for a given **scene address**, which corresponds to a specific instruction in the scene manager function SCENE_updateDisplay.

- scene_manager.c contains the function SCENE_updateDisplay which is a switch statement including a case for each available scene address, calling the appropriate function from any given module in lib/scenes.

- The active scene is tracked by an unsigned int exposed by scene_manager.h, which can be set by any function to the scene address for the desired transition scene.

SCENE_updateDisplay is called every loop of the main gameplay loop, after all game state updates take place. This redraws the display based on whatever constraints are provided in the currently active scene. Therefore, any given scene requires that it contain its own functionality to exit to other scenes.

Text-based selection menus are a special case in that they are all handled by a single menu generator, discussed in more detail in **2.4.6 Understanding The Menu Generators.** For anything more specialized a custom scene is required. Any custom scene added requires that a new scene address be defined and the case statement in SCENE_updateDisplay be updated to call the new primary function of your new scene.

3.4.6 Understanding the Menu Generators

In order to save on program overhead and simplify the codebase, a special scene defined in lib/scenes/menu_generator.c was created. This provides a simple scene function, SCENE_textMenu(), which requires the following arguments:

- target_LSTRING_HEADER, which should point to a menu header LSTRING from the localization files;
- target_LARRAY_OPTS, which is an pointer to an array of option display strings, again in the localization files;
- target_MARRAY, which is a pointer to a special array type (voidFuncPointer), which is an array of menu execution functions in the same order as the LARRAY_OPTS.
- An int target_count_opts, which is the length of target_MARRAY.

Based on these properties, the menu generation scene will handle input events and menu pagination. Text menus are fairly powerful. Since target_MARRAY is a pointer to an array of pointers to void functions, each entry in a menu can do effectively anything that can be done in a single void function, which is, effectively, anything, so long as the function is a void. In the simple case this may change the active scene. In more complicated cases, it may modify game state or hardware configuration in some way.

The best example of this functionality is the menu defined in scenes/menus/debug.c.

The title of any given menu (LSTRING_HEADER) and the names of the various options (LARRAY_OPTS) should be defined in a locale file. See **3.4.12 Understanding PETI Localization** for more details on the localization system.

3.4.7 Understanding Display State Management

Where the PETI development kit targets the MSP430FR5994 microcontroller, the available 8kb on-chip RAM is insufficient to individually store each individual pixel of the display (in the stock 128x128 pixel configuration). Because of this, managing display state relies on an abstraction method; text-mode display. This is a similar approach to the problem as was used in some of the early microcomputers like the Commodore VIC-20, though we’re taking advantage of greater processing power and can add some layers of sophistication not otherwise present.

All scenes share the same instance of a DisplayFrameNew struct called **DISPLAY_FRAME**. This structure consists of an array of special structures (DisplayLine) called frame, which fundamentally resembles one “frame” of screen animation; each contains two arrays of chars, one for the actual text to be displayed and one for directives to modify how the print is displayed (see below). A pointer to this object is called at the end of each scene for the DISPLAY_updatesOnly_enhanced function, together with a pointer to a **scene definition**.

In addition, each of the three print functions references a different font from font.h. In the same order as listed above, they use font16x16, font8x12, and font8x8. These fonts are further detailed in **3.5 Display Fonts Reference,** but in short form, 8x8 and 8x12 are two different sizes of textual fonts, whereas font16x16 contains display characters used to illustrate the PETI characters and gameplay elements.

At the end of each process of DISPLAY_updatesOnly_enhanced; the current DISPLAY_FRAME is copied into a second variable called PREVIOUS_FRAME. During each run of the function, DISPLAY_FRAME and PREVIOUS FRAME are compared, and only DisplayLines which differ between the two are written to the display. This is to remove a "scanline" effect that was present in earlier versions of the display code and to reduce complexity versus the legacy display code (see **3.4.7.1 Understanding the Legacy Display System**), overall optimizing both memory usage and processing time. An override flag is provided, FORCE_REFRESH, which is mostly an element of the legacy system, but remains available if needed. If FORCE_REFRESH evaluates to true, every DisplayLine will be written rather than only those that have changed since PREVIOUS_FRAME. This can be useful on the very first frame after SCENE_ACT has been changed.

3.4.7.1 Understanding the Legacy Display System

Note: This section details the "legacy" display system, which is the state of the display code prior to version v.0.0.6 of the firmware. It is slated for eventual deprecation when planned changes to the Debug Menu are made to allow the active species ID to be changed are implemented. Until then, the legacy system is only being used by the demo_mode.c scene. ***No new scenes should be created using this method.***

Operation remains largely textual, save that the unenhanced version of DISPLAY_updatesOnly is used, and that the frame passed into that function is a frame of dozens of pointers to arrays defined in demo_mode.c. This is extremely memory inefficient in both RAM and FRAM (due to the increased code complexity) and should be removed.

In particular, the DisplayFrame structure includes an integer pointer for each line of text that indicates whether that line should be printed. There is no frame-to-frame state management and scenes on the legacy system must determine *internally* whether each line of text needs to be redrawn.

3.4.8 Understanding the Alerts Subsystem

Note: The alerts system is only partially constructed as of version 0.2.0. Section **5.2 Alerts** details that implementation.

PETI is capable of raising notifications to the user in three main ways:

1. By raising a dedicated LED signal, either LED_ALERT or LED_BATT, with the latter being dedicated entirely to an as-yet-unimplemented Low Battery Warning Indicator;
2. Through the generation of an audio signal through the piezoelectric buzzer. As of version 0.2.0, this is implemented only as single tones of variable length, or;[1]
3. On screen, though this must be implemented at the scene level.

1 The intention is to provide a facility for stringing these together in sequence. It is probable that the current hardware design does not permit variable pitch.

The intention is that such alerts be used to fetch the player's attention when the pet has entered an undesirable condition.

3.4.9 Understanding the SPI Expansion Subsystem

The SPI expansion subsystem is not actually implemented as of firmware version 0.2.0. A discussion of the intended implementation is present on the project wiki:
https://github.com/ZAdamMac/PETI/wiki/Intended-Operation-of-the-SPI-Expansion-Subsystem

3.4.10 Understanding Onboard Power Management

The Onboard Power Management system is not fully implemented as of version 0.2.0 of the PETI firmware. We are presently relying on brownout protection built into the MSP430FR5994 itself. A discussion of the full power management considerations and their intended implementation is present on the project wiki:

A discussion of the intended implementation is present on the project wiki:
https://github.com/ZAdamMac/PETI/wiki/Intended-Operation-of-the-Power-Management-Utility

Beyond the intended facilities in using low power modes to extend battery life, input voltage regulation is provided between the battery and the 3v3 rail using a TPS60204 DC-to-DC converter. This holds the rail voltage at or near 3.3 volts regardless of the battery chemistry used, and provides the added facility of detecting low battery capacity and raising a digital alert, though handling of that alert is not yet implemented in firmware at the time of writing.

3.4.11 Understanding Our PRNG

PETI implements a limited-utility **Linear Congruental Generator** in game/entropy.c and exposes convenience functions and certain variables that other scenes would need through the associated header file. The RNG current state is stored as an unsigned integer, RNG_current_state. Of principal use to scene developers is the convenience function RNG_drawFloat() which returns a float based on the state of the RNG, bounded between 0 and 1.

Such a PRNG is considered to be of very low quality. While the version implemented as of v 0.2.0 of the firmware is stable and has a sufficiently long period and apparent entropy as to be unpredictable to a human, this is not a cryptographically secure random number generator and would not even be sufficient to use for gambling. It's a perfectly valid toy implementation, but should not be used as an example in more critical applications.

3.4.12 Understanding PETI Localization

Note: *This section discusses a feature of the PETI firmware that is not yet fully implemented, in the state that feature was in as of firmware version 0.0.7 and later. The intended operation is that the correct set of localization files be indicated by defining a symbol in the arguments to the compiler. However, the localization system headers are not currently in a state to support this.*

In order to facilitate localization of the PETI firmware; all human-readable strings are defined in one of several files in lib/locales. Each "locale" should be present in the library as a source file/header file pair, named <locale_code>_strings.c/h. The primary locale is Canadian English, provided for in en_CA_strings.c/h. Any new locale would need to contain all of the same variables as this locale pack.

Anywhere a scene would need a textual string that would be read and understood as *words* by a user, that string should be defined in the locale files and referenced as its variable name rather than defined absolutely in the scene itself.

In principle this would allow localization of the device to be changed by simply compiling the firmware with the appropriate localization pack. Given the size of FRAM, it may be possible in a future version to include ALL the locale files in FRAM and make the operating locale selectable from within the firmware itself, though this will likely require changes to the localization system design.

3.4.13 Understanding Human Input Handling

As of version 0.2.0, PETI uses a queue for handling human input. Each of the four input buttons A through D are configured such that they are handled by unique ports on the MSP430, and as a result a press of any of these buttons generates a unique interrupt. Each button's interrupt handler adds an integer to a queue before letting normal execution resume. The queue can hold up to eight input events.

At each screen refresh (that is, each time a given scene is called), an input handler function (SCENEKEY_handleInput) unique to that scene is called. It iterates over the inputs in the queue and removes them from the queue, and takes whatever actions are appropriate for the currently-active scene.

This was chosen in replacement to the previous method as it allowed for perceptually "smoother" scrolling of text menus.

4.0 Gameplay Reference

Current as of Firmware Version 0.2.0

When your game first turns on, you will be asked to input the current date and time. Cycle through the options using button C, and adjust the number value highlighted up and down using the A and B buttons, respectively. When you're done, press C until "SET" is highlighted, and then press C again.

4.1 Interacting with the Main Game Screen and Menus

Most of the time during gameplay, you will be looking at the main gameplay screen, which provides you with a playfield showing your delighful pet capering about, and a menu which is displayed across both the top and bottom parts of the screen.

You can move through this menu, starting with the top-left icon, by pressing the A button to go forwards through it, and the B button to go backward. Select the currently-highlighted option by pressing the C button, or clear the menu by pressing D.

The main game menu contains the following options:

- A scale icon, leading to the status menu (see **4.2 Monitoring Your Pet's Health**);
- A wrench icon, leading to the settings menu (see **4.9 Adjusting Device Settings**);
- A knife and fork icon, leading to the foods menu (see **4.3 Feeding Your Pet**);
- A smiling face icon, leading to the games menu (see **4.4 Entertaining Your Pet**);
- A bug icon, leading to the debug menu (see **4.10 Debug Menu Functionality**);
- A calendar icon, leading back to the calendar settings menu.

On many menus, you're going to be presented with just a series of text options. Icons on the screen will help you remember this, but you can move the selection bar (highlighted in black) up and down using the A and B buttons, select an option with C, and go back with D.

4.2 Monitoring Your Pet's Health

Your pet has several stats which can help you understand its health! Based on these stats, your pet may become irritable, get sick, or even change its evolutionary path. Be sure to keep an eye on your pet's stats and give it the care and attention it needs.

On the status menu, you can change pages with the A and B buttons. Pressing either C or D will take you back to the main game menu.

Your pet has the following stats:

- Hunger, which is a measure of how well fed they are.
- Fun, which is a measure of how well entertained they are.

- Discipline, which is a measure of their emotional and intellectual development.
- Weight, which is a numerical indicator that hints at their overall health.
- Age, which is the amount of time in days since they hatched from their egg, and;
- Special, which will show you some icons depending on special health properties.

Currently, as of the 0.2.0 firmware, Weight and Special are not implemented; the game will even tell you so!

Looking at Discipline, Hunger, and Fun, they are represented as bars. The idea is to keep the bar full (shaded). If your pet is particularly hungry, a warning icon (an exclamation mark inside a diamond) will appear. Be sure to feed or entertain your pet regularly to avoid this, as a pet that is very hungry or very bored for a long time is at increased risk of getting sick.

4.3 Feeding Your Pet

When entering the food menu, you'll be presented with two options:

- Food, meaning foods that primarily only affect hunger, and;
- Snacks, meaning foods that primarily (and sometimes only) affect snacks, and which may be unhealthy for your pet.

When selecting a food from the menu under either option, an animation will play where your pet eats it, and then you will return to the main menu, having fed your pet. Both the foods list and the snacks list can contain multiple options, so check regularly to see if anything new is available.

4.4 Entertaining Your Pet

A simple minigame is provided for the pet, in which you will play Rock, Paper, Scissors against your pet. Whichever of you gets the best score after five rounds wins (in the event of a draw, the player wins). On-screen prompts will provide you a reminder of which button means which choice and a simple tune will play throughout.

In the event that you win the minigame, your pet's "Fun" meter will be restored by 5 points, or about 1/3rd of its total size.

The minigame system is designed to allow developers to add new games easily, so keep checking in for updates!

4.5 Putting the Pet to Sleep

This feature is not yet implemented in the current version of the firmware.

4.6 Grooming Your Pet

This feature is not yet implemented in the current version of the firmware.

4.7 Disciplining Your Pet

This version is not yet implemented in the firmware.

4.8 Understanding Evolution

This feature is not yet implemented in the current version of the firmware. A discussion of the intended implementation is available in the wiki attached to the main repo for the project, here: https://github.com/ZAdamMac/PETI/wiki/Intended-Operation-of-the-Evolution-System

4.9 Adjusting Device Settings

While a menu option is provided on the main screen to adjust device settings, it currently just displays a message saying "Button Worked!", which was provided for testing the main game menu screen. Eventually, this will be a menu that allows you to enable or disable audio alerts and adjust other device settings.

4.10 Debug Menu Functionality

If Switch 2.2 is set to "on" (that's closed), the debug menu will appear at the end of the main game menu. Currently, it provides you with the following options:

- **Input Test**, which will display a message indicating that the "Button Worked!", which is a holdover from earlier, testing builds of the device.
- **Audio Check,** which contains commands that exercise the audio circuit in various ways.
- **Fill/Drain Hunger/Fun Bar** commands, to either fully fill or fully empty the respective bars.
- **Stage Select**, which allows the player to manually set the pet to any stage of life, including the Egg stage. This causes any other scene in the game to behave as though the pet has evolved to that state (because as far as the game is concerned, it has).
- **Entropy Check**, which is a small scene that displays current information about the output of the PRNG (entropy.c). Provided mostly for curiosity.

5.0 Per-Module Documentation

For a number of reasons, it is desirable to provide written documentation beyond simple code comments (or even docstrings) for each and every module that makes up part of PETI. While information conveyed in section 3 provided high-level overviews of various systems and interactions, this written documentation should provide more nuance, as well as contextualizing various code design decisions and limitations.

Due to the design of this service manual you are encouraged (if desirable) to print out the source code for each module and keep it in the same binder after each section of commentary for easy of paper reference, if not simply using a digital copy of the manual.

5.1 Main

Both the order in which hardware initialization functions are called, and the core gameplay/operation loop of the device, are defined in Main.C, which also contains the interrupt definition to handle the timer interrupts used to stop audio playback and wake the program roughly once per second.

Of particular interest is that VERSION is defined in main.h and this controls what version information is displayed to the user on the version screen.

Nothing should ever call Main or otherwise reference main.h unless for some reason the `VERSION` global is required.

5.2 Alerts

The Alerts system is in a rudamentary state as of version 0.2.0. In some ways this is because the use of alerts is not heavily desired at this early stage of development, as the game is not really “playable” and the alerts therefore serve as an annoyance more than anything.

However, the Alerts folder should contain the code defining any developer convenience functions to be called regarding either of the alert LEDs (the Alert LED **or** the Battery LED) or the beeper.

5.2.1 Alerts/Alerts

*Module Signifier Prefix: ALERTS_*

Alerts/Alerts defines a single function, ALERTS_hunger_fun_alert, which does very little. It first calls *BLINKENLIGHTS_raise* to set the ALERT LED to the ON state, then sends the signal *AUDIO_pulse(AUDIO_LONG_PULSE)* which starts the buzzer running and sets up Timer B0 for a long pulse. This is used by the game to signal that the hunger or fun stat has become critically low and is likely to be elaborated on in future versions.

5.2.2 Alerts/Audio

*Module Signifier Prefix: AUDIO_*

Alerts/Audio is to be used to store developer variables related to audio and the functions that consume them. It currently exposes a single function, *AUDIO_pulse*, which accepts an integer argument for the length of the desired time the buzzer should be switched on. This integer is then set up as the capture compare ceiling against a clock running at 125kHz, which means a value of 0xFFFF would cause the pulse to last slightly longer than half a second.

Two pulse values, AUDIO_LONG_PULSE and AUDIO_SHORT_PULSE, are also provided. They provide a delay of 0x4444 and 0x2222 respectively which was arrived at by experimentation as being aesthetically pleasing. The intended usage is to provide them as the argument to *AUDIO_pulse()*.

This code is legacy to a time when the audio system was frequency-pegged by using a tuned buzzer rather than a low-voltage speaker, and is likely to be significantly revised in the future.

Using timer interrupts in this way is desirable as it allows the calling function to simply send a signal that audio is desired and then the timer interrupt itself can handle stopping the audio output, preventing the execution speed of the scene that called the function from impacting the length or other properties of the audio output.

5.2.3 Alerts/Blinkenlights

*Module Signifier Prefix: BLINKENLIGHTS_*

Alerts/Blinkenlights exposes two functions presently which control the function of the “Alert” LED, on the back daughterboard of the development kit. *BLINKENLIGHTS_raise* and *BLINKENLIGHTS_lower* will turn on and off the LED, respectively.

The intended usage of this operation is that a scene or gameplay function that would raise the alert LED calls the function *BLINKENLIGHTS_raise*. The LED then remains lit indefinitely, until a confluence of the human interaction and main gameplay loop codebases detect that the player has pressed a button and call *BLINKENLIGHTS_lower*. This allows the LED to serve to alert the user to a condition if they happen to miss the audio signal, even if they return some time later.

5.3 Display

The display subsystem remains the single most resource intensive portion of PETI’s operation in terms of consumed cycles. If a sufficiently large amount of the screen is being redrawn on any given “frame” or “tick” of the main gameplay loop, it’s possible for this to take longer than the intended period of 500ms. When this happens, it appears as a noticeable slowdown in screen draw events to the player.

The display system itself endeavours to resolve this directly by both providing the most efficient possible usage of both FRAM and RAM memory, minimizing the number of clock cycles spent on display tasks, and exposing maximally-useful convenience functions to developers so that scene code remains comprehensible.

The core of this functionality is provided in Display/Display.

5.3.1 Display/Fontedit Source Files

The fontedit source files directory contains the .fnt font files managed by peti-helpers/fontedit, which are used to generate the font data in font.h.

These source files are provided for the convenience of other developers who may wish to implement custom characters or make changes to the font management.

5.3.2 Display/Display

*Module Signifier Prefix: DISPLAY_*

***Note:** Display/Display predates the version which introduced the mandate that all modules use clear prefixes on their function calls. Most functions intended for developer use have been refactored to include the prefix, but some and especially some symbols have not. Changes to the display module directly or indirectly impact basically every other file in the firmware and are undertaken only when absolutely necessary.*

***Note 2:** The following items were deprecated as of 0.0.6 (or potentially earlier) but have not yet actually been removed from the module as of 0.2.0:*

- DisplayFrame (struct) – replaced by DisplayFrameNew
- Display_UpdatesOnly (void func) – replaced by DISPLAY_updatesOnly_enhanced
- All printText functions that are exposed in the display.h file.

This module exposes a relatively enormous number of defined symbols, useful structures (and instantiated versions of those structures), and functions, all of which are used in developing Scenes. The following subsections will discuss these values in detail.

5.3.2.1 DISPLAY_FRAME (Global Structure Instance)

By including display, your module obtains access to DISPLAY_FRAME, an intended-global instance of the DisplayFrameNew structure. The structure is an array of DisplayLine structures, which each contain their own named arrays. The size of these arrays (in terms of their number of members) is determined at hardware initialization based on ***5.3.2.6 Resolution Marker Symbols*.** This ensures that they are created in such a way as to use the minimum viable amount of memory, which is important because at all times PETI actually maintains both the DISPLAY_FRAME and a copy of its previous state (conveniently known as PREVIOUS_FRAME).

Display Frame is the structure exposed to developers to let their scenes manipulate the display. As discussed in section **3.4.7** of this manual, the display is text mode. Each “line” of text is represented by the two members of the DisplayLine struct; `line`, which controls which caracters are displayed, and `directives`, which controls modifiers applied to those characters as they are displayed.

Individual characters can be addressed directly by thinking of them as cells on a grid: the index of `DISPLAY_FRAME.frame` sets the row and the index of either line or directives sets the column, as in the following example:

```
DISPLAY_frame.frame[0].line[0] = “A”;
```

This would set the top-left character on the screen to “A”. Depending on which font is being used this may or may not render as an actual A.

While line expects actual character input and directives could accept it, the directives are defined in such a way that they bitmask against each other and are additive. Each character in the directives array actually defines two different parameters:

- A font address (such as FONT_ADDR_0) telling later functions which font to use to translate between the characters in line and actual LCD state information, and;

- A “directive” (such as DIRECTIVE_REVERSED) that then performs an operation against the font data.

These two flags can simply be summed; i.e `FONT_ADDR_0 + DIRECTIVE_NORMAL` is a perfectly valid assignment. Each font size has up to sixteen possible font addresses (not all of which are defined). These are labelled sequentially as FONT_ADDR_0 through FONT_ADDR_F. Additionally, there are four valid directives:

SYMBOL	**Effect**
DIRECTIVE_NORMAL	Character, unmodified, is sent to the display
DIRECTIVE_REVERSED	The character is reversed right-to-left. This has no impact on the character order itself, just the way the individual character is displayed.
DIRECTIVE_NEGATIVE	The character is color-inverted compared to the normal display.
DIRECTIVE_REVERSED_NEGATIVE	Both reversal of the character and color inversion are applied

See the section **3.5 Font Reference** for information on which fonts are used at which font size, and what those fonts display.

Both of these members; the line member of the line and the directives member of the line, can also be assigned to with strcpy provided that the appropriate includes are made. This is extremely convenient when dealing with actual text.

The intended operation is therefore to loop/iterate/directly assign as needed to each line of DISPLAY_FRAME as needed and then to pass the pointer to DISPLAY_FRAME as part of a final call to DISPLAY_updatesOnly_enhanced and let that function perform the necessary “pruning” to figure out what lines need to be written.

Note: However, it is also desirable where possible to minimize the number of modifications to Display Frame if it’s known certain lines won’t be used, for the sake of respecting the limited computational performance of the device.. Scenes Manager exposes some useful variables that can be helpful in doing this.

5.3.2.2 DISPLAY_updatesOnly_enhanced (Void Function)

***Note:** It is likely that in some future version the `_enhanced` flag will be dropped from the name of this function. Its inclusion is only a temporary placeholder while certain unworthy-of-revision functions still rely on the deprecated DISPLAY_updatesOnly function as it currently exists.*

Provided a pointer to DISPLAY_FRAME (or, technically, any other DisplayFrameNew object) and an unsigned integer (intended to be one of the MODE integers, see below), this function will determine which of the universal change-only print functions to call, and pass the relevant data over. It also removes flags forcing a full rewrite of the entire screen (usually added by the scene manager). Indirectly, through its child functions, it iterates over each line in the DISPLAY_FRAME checking against PREVIOUS_FRAME for changes. If it finds the text or directives were changed, it then translates those changes into SPI bus traffic and writes them to the screen. Finally, when done, it copies the full argued-in DisplayFrame into PREVIOUS_FRAME, to allow the cycle to start over.

As of version 0.2.0, there are three modes defined:

MODE_XXX	Corresponding Scene Definition
MODE_GAME	SCENEDEF_game_mode
MODE_MENU	SCENEDEF_menu_mode
MODE_MINIGAME	SCENEDEF_minigames

These mode values control which lines of text are displayed at *which font size*. In some sense therefore, scene-definitions and font-address are *combined* to determine exactly which character is being referenced.

While as of 0.2.0 none of the currently-implemented scenes use this capability, in theory a scene could switch up which modes were being used between frames if desired. Also if desired, **5.3.4** has concrete expression of the meaning of these different scene definitions and the process for adding more.

5.3.2.3 DISPLAY_nthDigit (character function)

Previous to widespread adoption of string utilities in the codebase it was occasionally desirable to get the nth digit of an incoming integer. If for example you wanted the third-least significant digit of 1234, you would call this function in this way:

```
DISPLAY_nthDigit(3, 1234);
```

Which would, naturally, return 2.

This can be useful when constructing interfaces like that in the Calendar Set Menu and so has been retained.

5.3.2.4 LCD/SPI Function and Variables

There are a number of LCD and SPI specific functions set out in the display module, because they are there for the specific use of the display module. While they might have some limited utility to the developer, they are mostly used by either the display module itself or the hardware init module; perhaps at other times by the scene manager.

Of the non-deprecated subset of these functions, they are:

- Init_LCD, which defines the GPIO pin used as the LCD_CS signal (on both the expansion header and the actual LCD card) and sets up the SPI interfaces and VCOM timer.

- VCOM, which is used in SPI communication together with the function SPI_WriteLine. These are both deeply idiosyncratic to the physical MLCD module itself and so are contained here rather than in hwinit.

- MLCD_ prefixed variables, which are specific commands to the LCD module, and

- LCD Clear Display, which fully blanks the display and is only called at hardware init.

It’s rare that any scene developer should ever need to call any of these functions directly and they mostly exist so that largely static requirements of the health and function of the LCD module are maintained correctly by the main gameplay loop. For example VCOM is toggled once per second (or so) to prevent an unacceptable voltage differential building up in the LCD and limiting its service life or causing stuck pixels.

5.3.2.5 DisplaySplash (Void Function)

DisplaySplash is a convenience function. When called it blanks the screen and then sidesteps the usual character-based display process in order to write the contents of display/splash.h to the screen. This function is how the Arcana Labs bitmap (or arguably, any other appropriately-configured bitmap) is printed to the screen during the hwinit process.

5.3.2.6 Resolution Marker Symbols

Certain symbols were pre-defined to make life easier both for scene developers and for the display system overall when dealing with the possibility that someone might want to change the size of the display attached to PETI. SHARP makes compatable LCDs in other sizes and provided they have the same communication protocol (which is the current understanding), in theory one should be able to change their usage by changing only the PIXEL_X and PIXEL_Y values to the X and Y resolution of the new display.

As long as any new code that needs to know the screen size references those values, the firmware should adjust automatically once they are changed. This would, however, require custom firmware to be compiled by the user at each new screen size.

There are also FONT_SIZE_FLOOR_X and _Y which give the integer of the smallest font’s size in pixels. Currently, that’s a value of `8` for both. In principal if the `small` font size was replaced with an even smaller font or eliminated entirely these values could be updated to change the size of (among other things) DISPLAY_FRAME.

5.3.3 Display/Font

At time of the writing, all of the fonts are currently stored in this file as their exact output from the exporter from peti-helpers/fontedit. This is *highly* inefficient now that there can be multiple fonts for each font size.

The fonts themselves are arrays of arrays of character data, where the characters themselves encode to a bitmap of that full character, one “line” of the LCD display at a time. Expressed in binary, a 0 would indicate that the cell is black and a 1 would indicate that the cell is “white” (in other words, not lit). This is defined by the LCD manufacturer and can’t be easily changed.

While currently unsupported as of version 0.2.0, the planned changes to localization will likely also include small changes to font management to provide some ability for the localization file to either define which fonts are loaded or define a “default” font address other than FONT_ADDR_0. This is nuanced as a problem and not fully considered at this time.

All fonts need to be defined with the PERSISTENT pragma as they would otherwise exceed the amount of available RAM.

5.3.4 Display/Scene_Definitions

The Scene Definitions file exposes all defined scenes as special structures to be fed into printDeltas_universal. This allows for extreme portability in display layout while also divorcing that layout information from DISPLAY_FRAME.

At present, three scene types are defined:

MODE_XXX Code	**Arrangement**	**Image**
MODE_GAME	A medium lined, followed by six large lines, a small line (often unused), and another medium line.	
MODE_MENU	10 Medium Lines	
MODE_MINIGAME	8 Large Lines	

These line sizes control which of the printSmall, printMedium, or printLarge functions are called; in other words, which fonts will be relevant. Small refers to the 8x8 set of fonts, medium the 8x12, and large the 16x16. This is combined in the printout workflow with the FONT_ADDR_X portion of the directives information in order to decide which exact font is to be referenced.

By adding a new scene definition you have to do four things:

1. Add a persistent const SceneDefinition to the c source file with an appropriate name:
    1. Setting .rows as an array of {display_row, text_size} integer pairs
    2. Setting .lines_used to the number of lines used (i.e, the length of rows)
2. Add the same to the header file to expose it.
3. Modify DISPLAY_updatesOnly_enhanced to include case that points to your new display type.
4. Add a MODE_XXX flag to the display.h file so that scenes can reference this display mode.

For readability and to avoid magic integers as much as possible the text sizes have been enumerated to TEXT_SIZE_SMALL, TEXT_SIZE_MEDIUM, and TEXT_SIZE_LARGE.

5.3.5 Display/Splash

This header file simply contains (by default, anyway), the Arcana Labs Square-Iris-Eye Logo in the form of an appropriate bitmap. The bitmap is expressed in a top-left to bottom-right fashion. Monocrhome pixel arrangement was converted using the 0-for-black encoding required by the MLCD then converted to unsigned characters and stored as an array of those characters. This is stored, appropriately, as splash_bitmap and is stored as a PERSISTENT variable (that is, stored in FRAM). The splash bitmap was generated in a similar manner to the font data by literally encoding the state of each pixel as individual bits from top left corner to bottom right corner.

This method is absolutely not future proof and would break if changing the size of the display LCD by updating the PIXEL_X and PIXEL_Y values.

5.4 Game

The Game directory should contain those modules that directly relate to gameplay, such as collections of data used in scenes as reference information, helper functions, or the actual gameplay manager (which handles the flow of the gameplay state).

5.4.1 Game/Game Manager

The Game Manager contains a few special functions and exposes a few special objects which are required for the function of the game.

5.4.1.1 StateMachine (Public GameState Struct Instance)

Note: *This module hasn't had proper developer intention in several years. It's due for revision and still contains a large number of magic numbers and disregarding naming conventions.*

The most important object exposed by the Game Manager, from the perspective of both the operation of the Game Manager and all active scenes, is StateMachine, an instance of the GameState custom struct. StateMachine is the current gameplay state. It is, in essence, the closest thing in the entire codebase to an object representing the pet itself.

A Game State object has members:

- .AGE, an unsigned integer representing the number of calendar days passed since this pet hatched.
- .ACT, an unsigned integer setting the pet's current activity level:
  - 0 for Asleep (not implemented);
  - 1 for awake and healthy (default);
  - 2 for sick (not implement);
  - 3 for egg;
- .HUNGER_FUN, an unsigned int that represents two values as bitmasks:
  - Four upper bits expressing "hunger", and;
  - Four lower bits expressing "fun".
- .DISCIPLINE, an unsigned int which is intended to serve as a feedback into the evolution mechanism once implemented;
- .NAUGHTY, an unsigned int intended to be bounded 0-100 which represents the chance the pet alerts falsely (not implemented);
- .STAGE_ID, an unsigned int that aligns to the index on EVO_metaStruct and specifies which "species" the pet is currently evolved into, and;

- .HEALTH_BYTE, a yet-to-be-fully-specified unsigned int intended to hold bitmasked flags or nibbles containing information about the specific health of the pet, possibly including easter-egg state flags. This is currently planned to include the pet's weight variable but that isn't actually implemented.

In theory persisting the StateMachine instance of GameState to FRAM would provide tolerance to the pet for power loss, allowing a player to persist a playthrough of the game through a battery change. This is not currently implemented but is on the roadmap.

5.4.1.2 NEXT_STAGE_TRANSITION

Three global shared integers give the _AGE, _HOURS, and _MINUTES of the next time the evolution mechanism should be called; as evolution currently isn't actually implemented this effectively does nothing.

At the top of each minute according to the RTC, the program checks the StateMachine.AGE and the RCT hours and minutes against these stage transision variables to attempt to do a stage transition and call for evolution. Currently, this is only used outside of debug mode when hatching the egg (as the egg state is skipped when the device is powered on with debug mode enabled), but eventually will drive the core evolution loop.

5.4.1.3 Additional Functions

The Game manager has a handful of defined functions, which are mostly exposed so that they can be called by main. The first two, GAME_initStateStruct and GAME_evaluateTimedEvents are void functions accepting no arguments. Main calls both, the first before doing almost anything else just to set the StateMachine and check the state of the debug pin, before causing us to transition to the boot screen. The second is called once pre gameplay loop and we use it to work out events that run at a rate. For example, hunger decay data from EVO_metaStruct is checked to see if hunger needs to be deducted from the score. Ultimately, once implemented, all of the other "needs"-related functions like determining if the pet has fallen ill or is due for evolution will also be included here. As of 0.2.0, a number of these functions are provisioned for, but are commented out as they are not actually implemented.

Finally, a convenience function for scene developers called GAME_applyHungerFun is exposed. It accepts two signed ints as arguments, being the change in hunger and change in fun expected. If hardcoding a scene where the pet found a cupcake at random and ate it, you could simplistically implement this as:

```
GAME_applyHungerFun(1,15);
```

The function accepts both positive and negative values (say if you wanted a minigame that lead to the pet actually feeling hungry) and based on the input value does "smart rounding" to ensure neither value goes over or under the maximum before bitwise-arranging them back together.

5.4.2 Game/Entropy

The entropy module exposes quite a few variables and functions by the usual standards of PETI modules, which is unsurprising, because the intention is to use the Random Number Generator to introduce a fair amount of unpredictability into the game. This is done by implementing a Linear Congruential Generator with some reasonable hard-coded parameters to give a sufficiently large period (number of events before the pattern of the RNG repeats). The RNG is also being called by the main gameplay loop on every frame, meaning that the player will not necessarily know the current "stage" of that pattern they are in.

An unsigned integer RNG_current_state and signed integer RNG_session_seed provide some insight into the motion of the RNG for testing purposes. Neither are intended to be directly called. They are viewable on the device itself from the "Entropy Check" debug scene as "current entropy card" (converted to a float) and "initial seed" respectively.

There is a callable function RNG_getSeedWord used during gamplay initalization that extracts an unsigned-integer worth of data from the rather large TLV value hard-wired and unique to each MSP430FR5994 microcontroller, and returns it. Since we're grabbing a subset of this large integer every time the device powers on, in theory the seed value should change. As of the Revision C version of the development kit and version 0.2.0 of the firmware, this value only rarely changes. The value is selected using feedback from one of the on-board Analog Digital Converters. It is believed that this source is insufficiently noisy to provide this initial randomness, and conversations are ongoing surrounding whether or not it is necessary to correct for this.

RNG_initialize is a void function that accepts the output of RNG_getSeedWord after casting it to a signed integer. It saves this cast version of the seed to RNG_session_seed by assignment, and then sets RNG_current_state using the linear congruential formula on that session seed. The multiplier, carrier, and modulus of that formula are all modifiable by changing definitions early on in game/entropy.c.

The RNG_forceShuffle function is similar to initialize in that it also applies the formula, however it does so against RNG_current_state. This means that every time it is called the RNG advances by one. Most prominently, this function is used on the main gameplay loop to keep the RNG moving.

Finally, RNG_drawFloat is provided as a convenience function for developers attempting to integrate the RNG into their designs. Since RNG_current_state is being stored as an unsigned integer, it is not as useful for creating random effects. This function mirrors the functionality of other randomness functions in popular programming languages by returning a random floating-point number with the following rules involved:

- Force shuffle is called before the return value is calculated
- A bitwise AND function is used to make only the most significant byte of the RNG state relevant to the caluclation
- The result is then divided by (modulus – 1) to produce a float bounded roughly between 0 and 1.

The practical upshot of bounding the result between 0 and 1 is that it is relatively easy to implement a "percent chance" mentality when bracketing the results in your functions.

5.4.3 Game/Evo_data

evo_data's C source and Header File are at the core of the evolution system, which as of 0.2.0 firmware is not fully implemented, but is none the less core to the game's development. In a sense, it serves as a database of available "forms" for the pet to take as the evolution system progresses.

5.4.3.1 Stage Data Structure

The core feature of evo_data is the exposure of the custom structure type Stage. A stage is a collection of properties which define each stage of the PETI lifecycle and grants the following properties:

- \`int stage_id\`: An integer number giving the stage_id of the pet. It's actually mostly included for readability and should always match the index position of the stage object in EVO_metastruct (see below);
- \`int phase\`: a integer (magic integers as of version 0.2.0) which denotes the "lifecycle" stage the stage corresponds to. This allows for some functions to apply different effects at different life stages without having to use massive case statements individuating the stages on the same lines. The current valid values are:
    - \`0\` for the egg stage only;
    - \`1\` for baby stages;
    - \`2\` for "teen" stages;
    - \`3\` for "adult" stages;
    - \`4\` for "senior" stages.
- \`int size\`, one of \`EVO_size_small\`, \`EVO_size_med\` and \`EVO_size_large\` which are themselves defined in the header file. This is used to provide for the possibility of generalized solutions to drawing sprites per-frame. This correlates to the sprite size in the following way:
    - Small stages are a single text character;
    - Med stages are two characters wide and two characters tall;
    - Large stages are three characters square;
- \`const char animation\`: two dimensional array consisting of exactly two arrays of nine characters each. Each of the two arrays contains one "frame" of the default animation cycle for the stage. This is usually a walking/idle animation. Currently these are expected to correspond exactly to character codes on font16x16 which is the FONT_ADDR_0 of the large fonts. (NOTE: this addressing concern is shared for all char properties of Stage structs)
- \`char * faceRight\`: an array of arbitrary length giving a single static frame of the pet facing to the right. Directives can be used against this property to provide a facing-left animation.
- \`const char animationEating\` is an array with the same constraints as \`const char animation\` but which is intended to show the pet eating.

- `int rateHF`: an integer which controls the rate of decay in hunger and fun for the pet. This is defined such that the upper nibble of the number is used to determine the day rate in the fun score and the lower is used for hunger. Each nibble can be thought of as the number of points that should be removed from that score in a 15 minute period.
- `int highEvo`: a stage_id//index position that corresponds to the Stage this pet should be set to if the pet is the current stage, the evolution check is called, and the "good" condition for evolution is met. Note that as of firmware 0.2.0 this condition is not defined.
- `int lowEvo`: as high evo, but for if the "good" condition is not met.
- `int secretEvo`: an alternate value for evolution when a certain secret condition[1] is met.
- `int stageLength`: an integer number of days used to set the evolution delay when evolving into this stage. For example, if the stageLength value is 1, the game should wait 1 day before next calling the evolution function.

While this is a complete list of properties of this structure as of firmware version 0.2.0, it's not yet clear that the structure is necessarily complete. Notably absent is any provision for sleeping animations, differing states of hardiness per-Stage, or the use of multiple fonts to store character data.

5.4.4.2 EVO_metaStruct and EVO_count_stages

An arbitrary-length array of Stage structures, the EVO_metaStruct, is provided in the C file. This is a parent array of all of the Stage structures currently implemented, and is ordered by stage_id. Doing so in this way allows the developer to simply reference that structure and the current stage_id from the `StateMachine` global entity in order to get all of the necessary Stage data for whatever function they are implementing, pre-referenced to the appropriate stage for the current pet and game state. For sanity reasons, the length of this array is provided as an unsigned int, EVO_count_stages.

Note: As of version 0.2.0 there are undefined and poorly-defined values in this array. While all of the planned stages currently have their animation data present, other values like the xEvo mappings and rateHF are not necessarily set appropriately. This will be corrected in future versions as the functions that rely on that information get implemented.

5.4.4.3 EVO Sizes

The EVO size defintions can be (and are) imported by including evo_data.h into your module. Chances are this is convenient if you are writing something that is referencing this library and simplifies (e.g.) case statements around drawing the character sprite. These values are actually bound to the total number of characters that make up the sprite (which is why sprites are assumed to be square), rather than being purely sequential.

1 These secret conditions are not well documented for a very simple reason: the puzzle of discovering them for the player or a new developer is meant to be one of the design goals of PETI. An example of this that is especially easy to grasp is the intended path of using a 0Ω resistor to control debug mode in the "pocket" version of the device, instead of a clearly-marked switch.

5.4.5 Game/Food_data

The food data header and C file are set up almost exactly the same as the files for EVO data: they define two arrays of a specialty structure and the length of those arrays for development purposes, and the specialty structure contains the data needed to use that food in the game.

Unless new scenes are manually developed to use food data, modifying these two arrays (FOOD_array_foods and FOOD_array_snacks) will update the options available in the Feeding Menu – Food and Feeding Menu – Snacks submenus respectively. Instructions on doing this follow.

5.4.5.1 Understanding the Food Structure

Foods are defined into the array using a special structure typedef, `Food`. Each food has a number of properties:

- `unsigned int hunger_value`: A value which is expected to be between 0 and 15 which represents the increase in the hunger score when this food is consumed.
- `unsigned int fun_value`: As hunger_value, but impacting the "fun" score of the pet;
- `unsigned int mod_weight`: An integer value expected to be between 0 and 64 that would change the health byte of the pet on the StateMachine if the weight and health system were actually implemented. As of 0.2.0 this is not actually implemented but the property is provided as that implementation is planned.
- `char animation[3]`, a three-member array of characters which should represent the food being eaten. This is expected to be from the Large Font font16x16 (FONT_ADDR_0). Support for multiple fonts is intended for the future but not available as of version 0.2.0.[1]

The animation of the food is shown against the pet "munching" on it. In the provided example food "rice", animation[0] is a full bowl, animation[1] is the rice in the bowl partially cosnumed, and animation[2] is the empty bowl, to give an example of the intended use of the animation property.

5.4.5.2 Implementing a new Food

Implementing a new food type into the game is easy. By convention, foods which apply only to hunger belong in the foods array and foods which apply also to fun belong in the snacks array, but this is simply convention and the process is largely the same. It comes in a number of phases:

1. Add your necessary new icons to font16x16 using fontedit, and update the font in the source code.
2. Add a new food struct to the end of the appropriate FOOD_array.

1 It is extremely likely that this feature comes as early as version 0.3.0, which was the next planned `feature` release after 0.2.0. The font address system was added specifically because of how heavily constrained the "large" font is in terms of utilization and one of the best efficiencies in terms of effort-to-functionality would be to implement multifont support for foods first as a proof of concept before rolling it out to other animation functions.

3. Update your locales files (preferably across all locales) so that the need food is mentioned by name in either LARRAY_FOOD_FOOD_NAMES or LARRAY_FOOD_SNACK_NAMES, as matches your chosen array.
4. Update the value of FOOD_count_foods or FOOD_count_snacks as matches your chosen array to reflect the new length.

As long as all four of these steps are completed, on your next compile and flash the new foods should appear in the appropriate submenu without any need to modify the actual food menu or eating scenes. The mechanism for this is better explained in **5.7.3 Food Menu.**

5.5 Hardware Init

The hwinit directory, short for Hardware Initization, provides the custom library code for doing just that; initializing the hardware state of the device. This is different than initializing the game state of the device and should always be called beforehand, as it sets up the necessary harware timers and interrupt vectors to allow the rest of the system to function.

5.5.1 HWINIT/HWInit

The HWINIT.c/h module technically exposes four functions, all of which are called by main.c's main loop and none of which are of any real use to developers of new scenes or functions, but which may be useful in cases of expanding the device's overall functionality.

These functions are all unargued void functions:[1]

- Init_GPIO: through use of a large number of direct definitions, configure the actual pinout of the GPIO ports on the board. This function configures all four used human input buttons (see **2.1.1 Controls**) as well as configuration for the pinout of the buzzer, LED_ALERT and LED_BATT, P1.0 and P1.1. debug LEDs, and the pinout for the selector switches SW2.1 and SW2.2.
- Init_Timers: Sets up Timer A (specifically the capture compare register 0 on Timer A0) to serve as the master interrupt that controls the main gameplay loop. This also sets the system clock to 8MHz. The combination of the system clock speed and the TimerA configuration means that the interrupt for TimerA0CCR0 should raise about twice per second, providing the heartbeat for the main game loop.
- Init_RTC: sets the time on the RTC_C module to exactly midnight on the date and time provided by the DEFAULT_X series of defaults at the top of hwinit.c. These values all expect binary-coded-decimal values. These should usually be updated on version release to the day, month, and year of the last code change going into that version's release. DOW is slightly arbitrary. For our purposes, the convention is that a DOW value of "0x00" corresponds to sunday.
- Init_Watchdog: as a perfunctory matter, this disables the watchdog timer and clears any outstanding interrupts. This avoids certain issues where if the program failed to fully initialize in time the watchdog interrupt could get called and halt execution.

1 The large number of directly-called definitions from the driver library symbol set is a problem for the portability of the code and is the main reason it is difficult to compile this code for other MSP430 launch pads. In theory as long as the pinout is the same that should work. Future triage work is planned before the release of firmware 1.0 to replace this with a more flexible system.

5.5.6 HWINIT/Human Input

The human_input file pair were added in version 0.2.0’s development as a way to address deficiencies in older handling of human input events. The C file provides for the four interupt vector handlers that correspond to the four buttons of the daughterboard (see **2.1.1 Controls**). Each button happens to be on its own GPIO port, which simplified handling of the interrupts. When a given vector is called, we know what button raised the event and add items to a queue, HID_input_events_queue, which holds up to 8 items. There are also two unsigned ints, a ‘HID_interacted_flag’ and ‘HID_input_events_queue_depth’, which assist in input handling.

5.5.6.1 Building your Handler for HID_input_events_queue

By including the human_input header into your scene, you can fairly easily build your scene its necessary custom input handler as long as you keep the following facts in mind:

- The queue caps out at a length of 8 events.
- The queue is an array of integers which correspond to the BUTTON_X_PRESS symbols defined in the hwinit header file. Your handler will likely involve a switch/case statement that pivots based on those symbols.
- As you iterate over the HID_input_events_queue, you should continue to do so until you hit a BUTTON_NO_PRESS. This indicates you’ve reached the first position in the queue that doesn’t actually indicate a human input event.
    - An alternative is to reference HID_input events_queue_depth as the greater-than-or-equal-to target of a for loop, setting the event at each position to BUTTON_NO_PRESS as you go.
- When finished processing input, you should reset the queue by filling it with BUTTON_NO_PRESS and setting the HID_input_events_queue_depth to 0.

By convention, these handlers are defined at the scene level and are usually called SCENETAG_handleInputs. They are defined on a per-scene basis specifically because that gives scene developers the maximum flexibility in terms of how to handle inputs for a given scene, which is imagined to be especially useful for minigame development.

5.6 Locales

Note: As of version 0.2.0 the Locales system is a poorly-implemented stopgap destined for replacement. Currently, all localization references explicitly include `enCA_strings.c`. In a future version this will be replaced with a more generalized system of includes that will allow one of many localization header files to be compiled in, meaning that localization should be as technically simple as compiling with the intended language set.

However, for the purposes of this document, having even just one of these locale files exist for proof of concept allows the structure to be discussed. The locale files all share the same structure in terms of a number of LSTRING or LARRAY values. In the case of LARRAY values, they are themselves an array of strings. These strings are used in place of hard-coded strings in scene code in order to make localization easier. For example, the statistic name “Hunger” would be “Faim” under a frCA regime. Referencing the localization symbols rather than the specific strings means simply updating the used localization file should replace all instances of that word or phrase in the game with the correct version for your targeted language.

The table on the following page details in full the locale symbols available as of 0.2.0 firmware. However, this file is constantly changing. It is the standard practice to include a comment in the matched header file indicating which symbol is used for what.

Arrays are usually sensitive to the order of some other array. They are broken out seperately like this to specifically allow for ease-of-localization.

No individual string should be longer than 16 characters, which is the default PIXEL_X/PIXEL_FLOOR_X value.[1]

1 Future work around hardware ambiguation should remove this limitation as support for variable-size screens improves.

Table 5.6.1 Localization Objects

Symbol Name	What it is
LSTRING_HUNGER	The name of the hunger statistic
LSTRING_FUN	The name of the fun statistic
LSTRING_DISCIPLINE	The name of the discipline statistic
lSTRING_WEIGHT	The name of the weight statistic
LSTRING_AGE	The name of the age statistic.
LSTRING_SPECIAL	The name of the “special” statistic.
LSTRING_STATUS_HEADER	The header for the Status display menu
LSTRING_DEBUG_HEADER	The header for the debug menu
LSTRING_SOUND_CHECK	The header for the sound test menu
LSTRING_CALMENU_HEADER	The header for the calendar configuration menu
LSTRING_FEEDING_HEADER	The header for the feeding menu
LSTRING_FOOD_HEADER	The header for the feeding – food submenu
LSTRING_SNACKS_HEADER	The header for the feeding – snacks submenu
LSTRING_MINIGAME_HEADER	The header for the minigame selection menu
LSTRING_STAGE_SELECT	The header for the stage select menu
LSTRING_ENTROPY_INFO	The header for the RNG state-peeking debug menu
LSTRING_CURRENT_RNG	The name of the “RNG_current_state” value to a human
LSTRING_INITIAL_SEED	The name of the “RNG_session_seed” value to a human.
LSTRING_MINIGAME_PICK[1]	The header that appears at the top of the rock paper scissors minigame in attract mode.
LARRAY_DEBUG_OPTS	Array of human readable debugging submenus/scenes in the same order as they are set up in MENU_DEBUG_functions
LARRAY_DAYS_OF_WEEK	Array of the days of the week, starting on sunday, in human-readable three-letter abbreviations.[2]
LARRAY_EVO_STAGE_NAMES	Evolution stage names in the same order as EVO_metaStruct. These are included here rather than the Stage structure specifically to allow for ease of localization.
LARRAY_FOOD_TOP_MENU	Names of the food and snack submenus, in that order.
LARRAY_FOOD_FOOD_NAMES	Human-readable names of the foods in FOOD_array_foods, in order.
LARRAY_FOOD_SNACK_NAMES	Human-readable names of the foods in FOOD_array_snacks, in order.
LARRAY_SOUNDCHECK_OPTS	Human-readable names for the functions in the sound check array.
LARRAY_MINIGAME_TITLES	Human-readable titles for games in MINIGAME_array_games

1 This is likely a bad name for this value. Keep an eye out for changes to it in future versions.
2 The calendar setting menu and planned calendar display menu expect these to be exactly three characters each.

5.7 Menus

The menus directory contains paired header/source files which define the basic building blocks for various menus. The usage of these files is explained in more detail in **5.8.8 Menu Generator.** In short, the arrays and integers set up in these files follow the same general format and combine with LARRAYs and sometimes LSTRING_XXXX_HEADERs in the Locale files (see **5.6 Locales**) to power the menu-generator and render a menu when approrpiate.

These files always define, at minimum, a named array and a named count_options integer, which are usually marked as constants and given the PERSISTENT pragma (meaning they get compiled to FRAM). The count_options is always the length of the array and the named array should consist of \`voidFuncPointer\` objects (again, see **5.8.8 Menu Generator** for this definition) pointing to a function that should be run on each relevant position in the menu.

5.7.1 Debug Menu

The MENU_DEBUG_functions array performs the following functions, in order:

- Sends the player to the “button proofer” scene;
- Sends the player to the sound test menu;
- Sets the hunger value to 15, filling the bar;
- Sets the hunger value to 0, depleting the bar;
- Sets the fun value to 15, filling the bar;
- Sets the fun value to 0, depleting the bar;
- Sends the player to a “stage select” menu (special menu, see **5.8.10 Stage Selector**);
- Sends the player to **5.8.6 Entropy Debugger.**

The debug menu may not always be accessible. It requires the pin connected to SW2-2 (P3.6) to be pulled low, or it will not be rendered among the options in Main Game. This will be implemented differently in the “portable” version of the hardware.

5.7.2 Feeding Menu and Food/Snacks Menus

The feeding menu is extremely simplistic and simply switches the scene to either the Food or Snacks submenus.

The food and snacks menus each give arrays of specific foods and apply their effects by dropping into the Eating Animation with the appropriate arguments set.

5.7.4 Main Game

The main game menu is not actually rendered by the Menu Generator. It is instead treated by **5.8.1 Main Game**, but it works in a similar way as the other menu definition header files which is why it is included here and constructed this way.

Unlike the other menus described in **5.7**, this menu interpreter can only handle scene address changes. This is because no planned feature of PETI called for the application of effects directly from the main game standby screen; almost all features of the pet either involve entering a menu or, at minimum, playing a specific animation.

Instead, this file defines *two* arrays, MG_menu_icons and MG_menu_scene_addresses. These arrays are order-paired and should consist of the character code for a given menu option's display icon from font8x12 (medium font FONT_ADDR_0) and the corresponding SCENEADDR value to go to, respectively. An integer consisting of the length of these two arrays is also provided. **5.8.1 Main Game** uses these three objects to render the rows of icons at the top and bottom of the main game screen in a visual style consistent with legacy virtual pets, where in those previous designs the icons were rendered with custom LCD fields.

5.7.5 Minigames Menu

The Minigames menu links the scenes defined in the minigames subdirectory of scenes to the minigames menu scene; it's how those minigames are directly addressed by a player. It works in the usual fashion of calling VoidFuncPointers to set the scene addresses, which is slightly more convoluted than strictly necessary.

5.7.7 Sound Test

The sound test menu works by calling functions which directly call the AUDIO library in order to test out audio on PETI. As of firmware version 0.2.0 this only includes the short and long beep definitions. In future versions this will likely include a list of directly-callable “tunes” (sequences of pulse lengths) to call instead of or in addition to the pulses.

5.8 Scenes

From a human perspective, most of PETI's functionality is contained in scenes. By lines of code alone Scenes dominate the codebase, and since most PETI users are unlikely to ever look too deep beneath the hood, the scene system is how most of their interaction with the device takes place. Scenes govern all rendering on the display and provide access to functionality for gameplay, like minigames, feeding menus, and so on. While in gameplay-mechanical terms any new functionality will include changes to game_manager, in practice new scenes will also be required to handle that.

Key sections in this set of instructions will be **5.8.8 Menu Generator** and **5.8.9 Scenes Manager.** Understanding those two elements will be central to the process of developing new scenes for PETI.

5.8.1 Main Game

The main game screen is complex and provides three essential functions:

- A visual indicator that the game is still functioning, in the form of an idle animation of the pet exploring its small space.

- The rendering work needed to combine *metanimations* ***and*** *stage* data into that idle animation, and;

- presentation of the main game menu to the player, allowing them to access all other functions.

Main Game implements its own custom human-input rendering to allow you to:

- Press the A button to advance the cursor left-to-right across the top, then bottom, rows of the main menu.
- Press the B button to reverse the cursor
- Press the C button to change the active scene to the one corresponding to the selected icon, and;
- Press the D button to reset the cursor position to 0 and hide the cursor from the player.

It also unsurprisingly impelements a number of obscured functions that handle rendering metanimations.

5.8.1.1 Main Game Metanimations

Metanimations (a portmanteau of “meta” and “animation”) are special animation data that corresponds to the Stage.phase property of the current pet and which are defined in \`scenes/main_game/metanimations.h\`.[1] These are arrays of exactly 4[2] frames stored as the .d property of a Metanimation custom structure. Each frame is six strings of a width of eight characters, corresponding to the size of the central play area in the main game screen.

The metanimation format allows the developer to easily encode a pattern of movement on the screen for the character sprite in the form of custom directives in the string. Working in this way allows one metanimation to serve as the movement profile for all characters that share the same phase, saving on data by decoupling this behaviour from individual stages of life.

Each character on the row has a meaning, shown in the table on the following page. These meanings allow for semantic flipping of the animation and other operations which make the animations tolerant to various game conditions without adding bulk to the Stage animation data directly.

As noted in the included footnotes, this functionality is largely prototypical and should be seen as larval-state at best, and is likely to be replaced in whole or in part by version 1.0 of the firmware.

1 This has the unfortunate consequence of tying phase to size (because of the implementation of metanimations themselves) and also limiting the number of metanimations to phases. A change is being considered to add a special property to the Stage structure that gives a metanimation ID.

2 It would be trivial to allow arbitrary sizes with some small changes, which is also being considered.

Table 5.8.1.1 Metanimation Characters

Character	Effect
- (dash)	A blank space is printed
_ (underscore)	A blank space is printed with color in negative
0	The next character from the current species' animation's current frame is printed.
1	As case 0, but as though the full animation was reversed horizontally. This can cause problems when mixing with the 0 symbol and the two should never be combined.
2	As 0, but in inverted colour
3	As 1, but in inverted colour.

5.8.2 Boot Splash

The boot splash animation scene is actually extremely simplistic and is the first scene the player is exposed to when powering the device on or using the on-board BOR button.

It’s also very simplistic and uses very little of the usual display framework, relying instead on the SCENE_checkTransitionTimeCondition(); property from scene manager rather than counting frames.

In short, the scene calls display’s DisplaySplash function to print the display logo then prints the version string in small text directly across the bottom of the screen. None of this functionality relies on the usual DisplayFrame object, because DisplayFrame isn’t intended for handling bitmap graphical data.

5.8.3 Button Proofer

Buttonproofer is another “cop out” scene that sidesteps the display frame process to write directly on the screen. It was implemented early in development to test both the scenes system generally and the functionality of human input specifically, and direct prints the words “Button Worked!” on the screen after blanking it.

Button Proofer was officially deprecated in firmware version 0.1.0 and remains only because certain unimplemented menu items still point at button proofer instead of the scenes that will actually implement them. Removing the code prematurely will break complation.

5.8.4 Calendar-Set Menu

Currently the only human-facing interface for the Real Time Clock module is the calendar-set menu. It is displayed to the user immediately after passing through the boot screen and any time they select the calendar option from among the main menu options in main_game.[1]’

This scene leaves the RTC running (if it’s already running) but freezes a snapshot of the moment in time in the form of its initial state. A special menu is provided for the user in which they can set the clock one “digit” a time by:

- Using the A and B buttons to increment up and down, respectively,
- Pressing the C button to advance the cursor forward one position, and
- Pressing the D button to advance the cursor backwardone position.

When the SET option is highlighted, and button C is pressed, the time the user has input is committed to the RTC and the user is sent back to the main game screen.

The invitation to “enter the time” comes from the localization symbol LSTRING_CALMENU_HEADER. The “SET” button, however, is specific glyphs in the font8x12 font (which is medium font FONT_ADDR_0). This means that the SET button is not directly localizeable.

1 This is not a permanent access path and will eventually be replaced with a prettier real-time clock display scene instead, with access to the calendar setting menu being moved to an options sub-menu which does not exist as of 0.2.0 firmware.

5.8.5 Eating Animation

The eating animation is an interesting case in that it is purely an animation, in the sense there is no human input required and it will both enter and exit on its own. When a food is selected for consumption, that food is eaten in the eating animation. The scene handles animating the process of the food dropping down from above and being consumed by the pet, as well as applying the GAME_applyHungerFun() call with the appropriate values for the food it was called with. If any human input is provided, the scene skips to its final 10th frame of animation, applies the food's values to the pet's status, then sends you back to the main game screen.

5.8.6 Entropy Debugger

The entropy debugger is, like many other functions in the debug menu, simple. It draws a couple of labels on the screen and then shows the RNG_session_seed and RNG_current_state values; the latter by way of an RNG_drawFloat call.

This allows developers to monitor the health of the RNG directly, but isn’t useful for long-term analysis, and may be either reworked or removed in future releases.

5.8.7 Food Selector

The food selection scene is, at its most straightforward, a dedicated menu. In fact, it’s a dedicated subclass of the menu generator specifically for handling food events in a slightly nostandard way to the usual. This allows us to more or less ignore portions of the snacks_menu and food_menu code as well as generalizing the case of eating food. The Food selector sets a variable that is read by the Eating Animation to determine which food should be used, based on direct access to the food_data arrays.

5.8.8 Menu Generator

The Menu Generator header file only exposes two real areas of functionality:

1. The voidFuncPointer type, used to trivially address void functions as pointers, and;
2. SCENE_textMenu, a void function that is used as described in **3.4.6 Understanding the Menu Generators.**

Under the hood, the operation of menu generator.c operates in a more involved way. It implements custom input handling that is shared by all menus which use SCENE_textMenu, which should be the vast majority of implemented menus (the Food Selector menu and the Status Display “menu” are the obvious exceptions).

This facility also provides automatic pagination based on a single critical definition, MENU_active_lines. This needs to be set to the number of rows in the scene definition addressed by MODE_MENU (refer to **5.3.4 Scene Defintions**). If the scene is redefined, then MENU_active_lines also needs to be changed.[1]

1 This is only relevant when changing the size of the LCD display module in pixels, which is an activity that is outside of the main scope of this project.

5.8.9 Scenes Manager

The Scenes Manager is a critical piece of infrastructure that governs more or less all of data presentation to human users of the device. It is used to set up the mapping of all possible scene addresses, which are how PETI knows what scene to call (and therefore, what to draw on the screen) at each iteration of the Miain Program Loop. The module also exposes a large number of state variables that are used by scene developers as a shared state model in order to reduce memory overhead.

5.8.9.1 State Volatiles and Their Uses

A number of volatile variables, mostly unsigned integers, are provided for by the header file as described below. None have true default values, but a few have specified behaviours:

- SCENE_ACT is a global unsigned integer used to signal the current SCENADDR value that corresponds to the active screen. Using the analogy of reading a book, this is your current page number. A developer may signal for a scene transition by setting a new value to SCENE_ACT. On the next run of the main gameplay loop this will cause that scene to be evaluated.[1]
- SCENE_CURRENT_PAGE is a signed integer, which is necessary because of some of the available uses of it. The specific use in each scene is idiosyncratic to that scene. For example, the menu generator uses it to help with its pagination functionality, but the Rock Paper Scissors minigame uses it to determine what “phase” of gameplay you’re in.
- SCENE_CURSOR_POS, the other signed integer, is used to determine where the cursor is on a given scene that implements a menu functionality.
- SCENE_PAGE_COUNT, on scenes with pagination, determines the total number of pages in that menu. This is computed dynamically by the menu generated based on features of the input arrays when those generators are called. Some other scenes may of course use this value.
- SCENE_EXIT_FLAG should be set to any non-zero value when a condition has occurred that triggers the scene to exit. Most scenes defined as of 0.2.0 use this in one way or another for cleaning up before changing the active scene address.
- SCENE_FRAME is an unsigned integer intended to provide the exact count of the current frame of animation, usually since the scene began. This is useful for animations that do not loop infinitely (though even looped animations can be made using modular arithmetic against this number.

Of these flags, all but the SCENE_ACT are set to the value 0 when the SCENE_updateDisplay function detects that the SCENE_ACT value has been changed since the previous frame (this is done by storing SCENE_ACT in a value called PREVIOUS_FRAME after checking the value of PREVIOUS_FRAME.) This is discussed in more detail in **5.8.9.3 Scene Update Display.**

1 This is very important. Changing SCENE_ACT has no impact on the frame that is currently being drawn. When a player does an action or a condition is met that changes SCENE_ACT, the current frame will still be drawn and the scene will only change (from the player’s perspective) on the next run of the loop.

5.8.9.2 Scene Address References

As of version 0.2.0, the scene address (SCENEADDR) values are manually defined one at a time based on a largely arbitrary arrangement of their values. The specific values do not actually matter and can be changed at will, as long as the SCENEADDR symbols themselves stay the same. These symbols are directly referenced both by scenes that define scene transitions and by SCENE_updateDisplay.

It is mandatory from firmware version 0.0.7 onward to include a comment alongside each of these defines being explicit about where the address goes.

5.8.9.3 Scene Update Display

The core of the scene manager is the function SCENE_updateDisplay. This is an unargumented void function which performs the following actions in the following way:

1. Checks if SCENE_ACT is a different value than PREVIOUS_SCENE. If so, it calls the display function LCDClearDisplay to blank the display and sets the FORCE_REFRESH flag, which causes every single line of DISPLAY_FRAME to be drawn this time around. As mentioned previously in this section it also resets most of the SCENE state integers to 0.
2. Sets PREVIOUS_SCENE to SCENE_ACT
3. Uses a large switch/case statement, switching on SCENE_ACT with each possible address as a case, and calls that scene's function, along with any arguments necessary.

When developing new scenes, in addition to defining your source code, you must edit the switch/case statement in SCENE_updateDisplay to add a new case with a unique SCENEADDR (which you must also add as a define statement to the header file). You can include any code you want in the case statement, though by convention this should usually be a single call to a single function, followed by the mandatory break statement. This function should serve as a wrapper for all the necessary complexities of your scene, which itself should be defined and handled in your code (unless you are simply invoking one of the generator scenes, like SCENE_TextMenu.

5.8.9.4 Scene Transition Timer Conditions

Though rarely used[1], the scene manager also has a facility to set a non-interrupt flag called the Scene Transition Time, stored as the unsigned integers SCENE_TRANS_SECONDS and SCENE_TRANS_MINUTES. These values are set by calling SCECNE_setTransitionTimeCondition() which expects an integer number of seconds as its argument. The function then turns your integer into the wall-clock integers for Minutes and Seconds respectively that your time delay will have elapsed by, based on the state of the configured Real Time Clock.

The companion function SCENE_checkTransitionTimeCondition then compares the current wall-clock time according to the RTC with the two variables mentioned above. If it determines that the transition time is in the future, it returns zero. Otherwise, it returns a non-zero integer. This is different from the "timed events" considered by the GAME_evaluateTimedEvents function and is intended to be used in

1 Currently used only by the egg functionality, as of v0.2.0.

specific scenes. An example use case would be causing a scene to “time out” if left idle for too long, possibly triggering a transition back to the main game screen.

5.8.10 Status Menu

The status menu implements a three-page custom menu which displays key statistics about the pet's current state to the player. Most importantly for day to day use, it translates the HUNGER_FUN byte into a pair of bars, each showing the relative state of the hunger or fun statistic. Ultimately, it will also show the pet's current weight and any special condition flags, though as of version 0.2.0 of the firmware, these features are not implemented.

The pet's age is also rendered; however at present it is done in a slightly archaic way that does not provide for easy localization. Fixing this in a future version will be a relatively trivial task.

5.8.10 Stage Selector

Stage select implements a custom variation of the TextMenu codebase that is designed specificaly to accept the EVO_metaStruct as its list of available options. Varying the menu generator in this way meant that instead of writing a unique voidFuncPointer for each possible pet state, the menu can read in the EVO_metaStruct itself and simply use a bit of multiplication to figure out the desired pet Id for a given stage before referencing that pet's STAGE_ID and setting that to the active stage ID on the StateMachine. This causes the pet to instantly evolve to whatever state without transitioning or showing any evolution animations and is intended for testing only, which is why it exists in the debug menu rather than an options menu.

6.0 Bill of Materials

In addition to the two parts referenced in section **2.4.1**, the following components are needed to fully populate the development kit. A bill of materials relevant to your purchase situation was included with your order if you purchased the kit from Arcana Labs. For the purposes of this manual, a full listing of commodity components is included. For the most part these parts are commodity, jellybean parts. Depending on the time since this document was last updated, you may have to make suitable substitutions.

6.1 Control Board Bill of Materials

Position	Description	Mfg. P.No.	Note
J1, J2	Pin Header Female 2x10, 2.54mm pitch	RS2-20-G	Commodity Part, Any manufacturer will do
C1, C2, C3, C4	Ceramic Capacitor, 10 nF	C324C103K3G5TA	Expect 5mm pin pitch
SW1, S2, S3, S4	Momentary Pushbutton Switch, 6mm, SPST-NO	1825910-6	450-1650-ND

6.2 Rear Expansion Board BOM

Position	Description	Mfg. P. No.	Note
J1, J2	2x10 Male Pin Headers, 2.54mm pitch	61302021121	Manufacturing from post strip not recommended; requires careful alignment for good fitment with launchpad.
EXP_SPI	6 Pin right-angle connector, 2.5mm pitch	PH1RB-07-UA	Cut one pin off if using suggested part
LCD_CS			Reuse above offcut.
D1, D2	3mm Diameter 3-5v LED, Colored	151031VS06000	If possible provide two different colors.
C1, C2	Ceramic Capactior, 1 uF		Commodity component, expect 5mm pin pitch
C3	Ceramic Capacitor, 2.2 uF		Commodity Component, expect 5mm pin pitch
R1, R2	7.62mm Axial Resistor 100 ohm		
R3	7.62mm Axial Resistor 100 kOhm		
R4	7.62mm Axial Resistor 649 kOhm		
R5	7.62mm Axial Resistor 340 kOhm		
LS1	Magnetic Buzzer, 3V, 11mm	WST-1208T	
SW1	DIP 3 position switch		Revision D Only
SW 1, SW2	DIP 2 Position Switch		Revision C Only
U1	3.3V Voltage Regulator/Charge Pump IC	TPS60204	Similar 60205 model NOT COMPATIBLE
BT1	Battery Holder, 2xAA Series Circuit	2462	