Two Player Game System for PyGamer and RFM69HCW Radio Wing

Created by Chris Young


Last updated on 2023-08-29 04:28:51 PM EDT
# Table of Contents

## Overview
- Hardware Required
- Skills Required

## Assembly and Set Up
- Check Out Other Learning Guides
- Wiring the RFM69HCW FeatherWing
- Installing The Software
- Updating Firmware
- Loading Sound Effects
- Testing Your Equipment

## Demonstration Games
- Uploading Games
- Basic Game Play Procedures
- Accessibility Features

## Playing Tic-Tac-Toe

## Playing Battleship
- Using Sound Effects
- Manual Ship Placement
- Game Play

## Creating Your Own Games
- Overview
- Your Main Program

### The basePacket object class
- How Packets Are Transmitted

### The baseMove object class
- Creating a Derived Move Class

### The baseResults object class
- Creating a Derived Results Class

### The baseGame object class

### The baseRadio object class
- Our RF69Radio object class

## Other Support Software
Overview

The Adafruit PyGamer is a powerful platform for creating your own handheld games. It can be programmed in MakeCode, CircuitPython, and Arduino C++. However most of the games that have been developed for this platform have been single player games. In this guide, learn how you can attach two RF69HCW FeatherWings to the back of two PyGamer devices and use RF radio communication to construct two player games.

It's all made possible by the TwoPlayerGame Library written in C++ using the Arduino IDE. The library provides you with base object classes that you can extend with your own custom classes that define the rules of your game. The base classes form a game engine that handles negotiation of the start of the game between the two devices, the back-and-forth transmission of moves and responses, and other housekeeping duties. This lets you focus all of your efforts on creating the game itself.

This includes some sample demonstration games "Tic-Tac-Toe" and "Battleship" that you can use as models for creating your own two player games.

The object oriented modular design of the game engine will also allow us to develop other types of two-way communication such as Bluetooth, infrared or other RF systems without needing to modify your game at all. You can simply drop in a different communication module and recompile. These other communication systems are still a work in progress, but the RF69HCW FeatherWing is fully supported now.

The sample programs provided here also run on the PyBadge which is not quite as feature-rich as the PyGamer. Notes are provided of any differences throughout the tutorial. It will not work with the PyBadge LC, a low-cost version, because it does not have a socket on the back to plug in an RF board.
The system should also be compatible with the Adafruit Clue board but will require implementation of a different communication method, using Bluetooth instead of the RFM69HCW 'Wings.

**Hardware Required**

To implement this project, you will need 2 PyGamer M4 Express units and 2 RFM69HCW Radio FeatherWings.

**Adafruit PyGamer for MakeCode Arcade, CircuitPython or Arduino**
What fits in your pocket, is fully Open Source, and can run CircuitPython, MakeCode Arcade or Arduino games you write yourself? That's right, it's the Adafruit...
https://www.adafruit.com/product/4242

**Adafruit Radio FeatherWing - RFM69HCW 900MHz**
Add short-hop wireless to your Feather with these RadioFruit Featherwings. These add-ons for any Feather board will let you integrate packetized radio (with the RFM69 radio) or LoRa...
https://www.adafruit.com/product/3229

Note American users should order the 900 MHz model and European users should use the 433 MHz model.
Adafruit Radio FeatherWing - RFM69HCW 433MHz
Add short-hop wireless to your Feather with these RadioFruit Featherwings. These add-ons for any Feather board will let you integrate packetized radio (with the RFM69 radio) or LoRa...
https://www.adafruit.com/product/3230

If you are going to use sound effects, you will also want to add a speaker or use headphones in the provided stereo headphone jack. Caps for your buttons and an acrylic case are nice additions as well. You will also want a Lipo battery for portable use. Or get the whole bundle in the PyGamer Starter Kit.

Adafruit PyGamer Starter Kit
Please note: you may get a royal blue or purple case with your starter kit (they're both lovely colors) What fits in your pocket, is fully Open...
https://www.adafruit.com/product/4277

Adafruit PyGamer Acrylic Enclosure Kit
You've got your PyGamer, and you're ready to start jammin' on your favorite arcade games. You gaze adoringly at the charming silkscreen designed by Ada-friend...
https://www.adafruit.com/product/4238
Mini Oval Speaker with Short Wires - 8 Ohm 1 Watt
Hear the good news! This wee speaker is a great addition to any audio project where you need 8 ohm impedance and 1W or less of power. We particularly like...
https://www.adafruit.com/product/4227

Plastic Button Caps For Square Top (10-pack) - 8mm Diameter
These Reese's Piece's lookin' bits fit perfectly on top of tactile buttons with 2.4mm square tops and give a satisfying 8mm diameter surface area for your fingers to...
https://www.adafruit.com/product/4228

Lithium Ion Polymer Battery with Short Cable - 3.7V 350mAh
Lithium-ion polymer (also known as 'lipo' or 'lipoly') batteries are thin, light, and powerful. The output ranges from 4.2V when completely charged to 3.7V. This...
https://www.adafruit.com/product/4237

As mentioned earlier, the system will also work with the PyBadge as an alternative to the PyGamer but not with the PyBadge LC. NOTE that the PyBadge does not have a headphone jack, so if you want to use sound effects you will need a speaker.
Adafruit PyBadge for MakeCode Arcade, CircuitPython, or Arduino
What's the size of a credit card and can run CircuitPython, MakeCode Arcade or Arduino? That's right, it's the Adafruit PyBadge! We wanted to see how much we...
https://www.adafruit.com/product/4200

Adafruit PyBadge Starter Kit
What's the size of a credit card and can run CircuitPython, MakeCode Arcade or Arduino? That's right, it's the Adafruit PyBadge! We wanted to see...
https://www.adafruit.com/product/4623

Skills Required

This is a relatively simple project if all you want to do is play the games that are provided. This tutorial may be updated if new games are added to the GitHub repository. You will have to solder a set of header pins on the RFM69HCW and solder an antenna and some jumper wires. There are a couple of flags you will need to edit in the code before uploading. However no programming skill is necessary.

If you want to develop your own games using the TwoPlayerGame Library, then it will take a reasonable amount of knowledge of object oriented programming techniques in C++. The detailed descriptions of the object classes and methods might serve as a lesson in object oriented programming techniques for a programming student.
Assembly and Set Up

Check Out Other Learning Guides

We suggest before you begin that you take a look at the learning guides:

- PyGamer Learning Guide
- PyBadge Learning Guide
- RFM69HCW Radio FeatherWing Learning Guide

These guides will show you how to add a speaker, battery, and button covers to the PyGamer or PyBadge. One of the demo games, Battleship, makes use of sound effects, so you will need to install a speaker or insert a pair of headphones into the headphone jack of the PyGamer. PyBadge does not have a headphone jack, so if you want to use sound effects you will have to install the speaker.

The other tutorial will instruct you on wiring jumpers and an antenna to the RFM69HCW. Also see the information about wiring below.

Wiring the RFM69HCW FeatherWing

The radio FeatherWing requires 3 jumper wires to be installed. The learning guide for this board does not specify how to wire it for a SAMD51 M4 processor such as used in the PyGamer/PyBadge. This guide uses the pins recommended for a Feather M0 and it works quite well.

Specifically the "CS" pad should be wired to the "B" pad. The "RST" pad should be wired to the "A" pad and the "IRQ" pad should be wired to the "D" pad. See the image below.
This is the recommended wiring for an M0 according to the learning guide for this device but it does not match the example programs in the RadioHead Library. If you want to run any of their examples or test programs, you will have to modify the code as described in the section "Installing the Software" below. Included in this guide are some test programs pre-configured for this project as well.

Additionally you will need some sort of antenna. I used a 3 inch long piece of wire as a whip antenna, but there are other antenna options you can use. For details see the learning guide Radio FeatherWing in the section "Assembly ()".

You will need to solder on a set of header pins. We recommend you do that after you have installed the jumpers.

When completed, insert the Radio FeatherWing into the back of your PyGamer or PyBadge.

Installing The Software

This project relies on three large libraries of code. The PyGamer/PyBadge uses the Adafruit_Arcada_Library and numerous other libraries that it depends upon. If you download that library using the library manager of your Arduino IDE it will automatically download any dependencies as well. For more details on installing the Adafruit_Arcada_Library see the learning guide linked here ().

Additionally the RFM69HCW FeatherWing relies on a third party library called RadioHead. For complete instructions on how to install this library refer to the learning guide linked here ().

Finally we need the TwoPlayerGame library which is available open source on my GitHub repository. It will have to be installed manually.

Here is a link to the GitHub page for this library. The green button below contains a link to download a zip file directly.

https://github.com/cyborg5/TwoPlayerGame ()

Download the TwoPlayerGame Library in Zip format from GitHub

Unzip the archive into your Arduino/libraries folder and rename it from TwoPlayerGame_Master to just TwoPlayerGame. For further information on manual installation of libraries into the Arduino IDE, check out the guide linked here ().
We have provided some demo software in the TwoPlayerGame/examples folder. Among them are some test programs which are preconfigured transmit and receive demos for the Radio FeatherWing.

If you want to use any of the test programs that come with the RadioHead library you can find them in the Arduino/libraries/RadioHead/examples/feather folder. Note however that the default wiring used in those sample programs does not match the ones that we use. They do not have an option for the PyGamer SAMD51 M4 processor. You can modify their example programs as follows.

Look for the section of code that looks like this...

```c
#if defined(ADAFRUIT_FEATHER_M0) // Feather M0 w/Radio
#define RFM69_CS      8
#define RFM69_INT     3
#define RFM69_RST     4
#define LED           13
#endif
```

Modify that section so that it reads as follows...

```c
#if (defined(ADAFRUIT_FEATHER_M0) || defined(ADAFRUIT_PYGAMER_M4_EXPRESS) )
#define LED           13
#define RFM69_CS      10   // "B"
#define RFM69_RST     11   // "A"
#define RFM69_INT     6    // "D"
#define RFM69_IRQN    digitalPinToInterrupt(RFM69_INT)
#endif
```

Updating Firmware

Can you should make sure that you have the latest bootloader and CircuitPython distribution for your device. Click on the appropriate link below for your device.

- PyGamer Download Page ()
- PyBadge Download Page ()

At the bottom of the page you will see "UF2 Bootloader". Click on the "DOWNLOAD UPDATER UF2" button. This will download a file such as update-bootloader-arcade_pygamer-v3.10.0.uf2 or update-bootloader-arcade_pybadge-v3.10.0.uf2 although the version number may be different.

You should double-click the reset button on your device and the PYGAMERBOOT or PYBADGEBOOT drive will appear on your computer. Drag-and-drop the .uf2 file onto that drive.
In the next section where it talks about loading sound effects onto internal QSPI memory, it mentions that when you upload the drive.ino program it should automatically go into CircuitPython mode. If for some reason it doesn't, you can force it into that mode by dragging and dropping the latest CircuitPython .UF2 file. So while you are on the download page, go ahead and get the latest CircuitPython at the top right of the page where it says "CircuitPython". The files will be named something like adafruit-circuitpython-pygamer-en_US-5.3.0.uf2 or adafruit-circuitpython-pybadge-en_US-5.3.0.uf2 although the version number may be different.

Loading Sound Effects

The Battleship sample program makes use of sound effects in ".wav" format. You can load them onto the internal QSPI memory of the PyGamer or PyBadge. The PyGamer has 8 MB of internal memory while the PyBadge has only 2 MB. The initial design of the game used sound files that were too big for the PyBadge so we have provided a separate set of striped down mono low sample rate sound effects for the PyBadge. If 8 MB is not enough for your own games, you can also use a microSD card inserted into the slot in the back of the PyGamer. The PyBadge does not have an SD card slot.

If you are using an SD card and have an SD card reader on your computer you can simply insert a formatted SD card in your computer, drag the appropriate file onto it, eject the card and place it in your PyGamer.

Alternatively you can use a program we supply to turn your device into a virtual disk drive and drag-and-drop files directly onto it. If you are using the internal memory, this program is the only way to load sound files onto your device and the procedure is slightly different for either. If using the virtual drive feature, format the SD card and insert it into your PyGamer before turning it on or connecting it to your computer for uploading.

Find the sample program drive.ino in the TwoPlayerGame/utilities/ folder and load it into the Arduino IDE. It will turn your device into a virtual disk drive so that you can drag-and-drop the sound files into it. There is a minor modification that you need to make to the program depending on whether you are using the SD card or the internal QSPI memory. Look for the line that reads:

```
define USE_SD_CARD false
```

If you are using the internal memory then leave it `false` and if you are using an external SD card edit it to be `true`. 
When uploading this program, you must change the USB Stack on the Arduino IDE from "Arduino" to "TinyUSB". You will find this option under the "Tools" menu as seen below.

NOTE: the drive.ino MUST be uploaded using TinyUSB mode but all other programs in this tutorial will work with either TinyUSB or Arduino stacks.

If you are using the internal memory, after uploading drive.ino you should see a new drive on your computer named CIRCUITPY. If for some reason you don't, you will have to force your device into CircuitPython mode as follows:

Double-click the reset button on your device and you should see a drive appear on your computer named either PYGAMERBOOT or PYBADGEBOOT. Drag-and-drop the Circuit Python .uf2 file that is appropriate for your device. The drive should change to CIRCUITPY.

Once you see CIRCUITPY you should then backup any files that might be on there in case you need them for something else. After that, erase all of the files from your device and create a folder in the root directory called /wav. Then drag-and-drop the appropriate sound files into that folder.

If you are using an external SD card and want to use the drive program, make sure you have edited the `USE_SD_CARD true` as described above. Connect the device to your computer with a USB data cable. You do not need to be in CircuitPython mode before uploading. Upload the drive.ino program and you will see a drive on your computer name "USB drive". Create a folder in the root directory called /wav. Then drag-and-drop the appropriate sound files into that folder.
Copy the sound files for the PyGamer from the TwoPlayerGame/sounds/battleship/PyGamer folder. These can be used on either the internal memory or the SD card.

If you are using the PyBadge you will need the smaller versions of the sound files to copy to the internal memory. Copy them from the TwoPlayerGame/sounds/battleship/PyBadge folder.

NOTE that the Tic-Tac-Toe game does not use sound effects. We may add other games to this repository in the future and we will provide sound effects for them in the TwoPlayerGame/sounds folder.

Testing Your Equipment

We recommend that you run some of the example programs for your PyGamer or PyBadge which can be found in the Adafruit_Arcada_Library/examples folder.

There is a example program called the sound_test.ino in the TwoPlayerGame/utilities folder that you can upload to test your sound effects. You should edit its `USE_SD_CARD` flag to `true` or `false` as necessary before uploading it.

Also in the TwoGamePlayer/utilities folder look for Demo_RX.ino and Demo_TX.ino. Upload one of them to one of your devices and the other to the other device to test transmit and receive functions. These are pre-configured versions of example programs in the RadioHead library.

Once you are certain everything is working, move onto the section on "Demonstration Games" and try out one of our games such as Tic-Tac-Toe or Battleship. That section will instruct you how to upload the games so that your two devices will be configured properly to talk to one another.

Demonstration Games

We have provided 2 demonstration games. We wanted something very simple that would illustrate how the TwoPlayerGame engine works so we implemented a simple "Tic-Tac-Toe" game. Then to really illustrate the power of the system, we implemented a classic game of "Battleship".

The PyGamer display alternates between a view of your ships and a radar screen of your enemy territory. You select a location on a 10 x 10 grid and fire a shot. The other device will send a signal back telling you if it was a "hit" or "miss". There are added sound effects for this game including cannon fire, hit and miss sound effects, and
voice clips saying such things as "You stank my battleship". There is a third
demonstration game myDemoDebug.ino that we used during the development of the
game engine that is completely text-based. It transmits messages back and forth and
displays the results on the serial monitor as well as taking input from the serial
monitor. It could be a useful starting point for creating your own games.

Uploading Games

You will have to make a slight modification to the software before uploading it to your
two devices.

On any of our games at the top of the main page you will see the following line:

```
#define IS_PLAYER_1 true
```

You should compile and upload the game to one of your devices with that definition `true`. Then then modify it to read `false` and upload it to the other device. This single
`#define` statement is the only difference between the software on the two devices.
It sets up addresses for the packet radio system so that the radios can talk to one
another. If you don't make this modification then they both try to communicate as
address "1". If you make the change them one of them will have address "1" and the
other address "2".

Basic Game Play Procedures

All of our demonstration games and likely any game created for this system will have
a basic standard flow of the game as follows.

When you first power on your device or press reset or upload the code, you will see
an opening splash screen and then your device will begin transmitting a signal
offering a game to another device. After a few seconds if it does not find anyone to
take its offer, then it goes into seeking mode which means it's seeking an invitation
from another device. It will stay in that mode until it finds an offer or you press reset to
start over.

If you turn on or reset both devices simultaneously, it's likely that they would both be
offering at the same time and neither one would be accepting. So you and your game
partner will have to ensure that you start the system one or two seconds apart from
each other.
Once the connection has been made, the offering device will perform a virtual coin toss to decide who goes first. Both devices will be informed of the results of the toss and gameplay will continue from there. The winner of the toss will make a selection by moving the cursor around using the joystick and then will select a location using the "Select" button. Your move will be transmitted to the other device, the results of that move if any will be transmitted back to you and then the other player will take their turn.

**Accessibility Features**

As a result of my disability I'm unable to operate the joystick or push buttons on the PyGamer or PyBadge devices. Therefore I implemented an alternative input method. In both battleship.h and tictactoe.h look for the definition...

```c
#define ACCESSIBLE_INPUT false
```

If you edit this to `true`, the program will use an extended version of the Adafruit_Arcada_Library that allows simulation of the joystick and button pushes by typing commands into the serial monitor. You can enter "U", "D", "L", or "R" for moving the joystick Up, Down, Left, or Right respectively. Entering "A" or "B" simulates pressing those buttons. Entering "E" for "enter" simulates the "Select" button while "S" simulates the "Start" button. The letters may be either upper or lower case. You will have to type the command letter and press Enter on your keyboard after each command.

All of the development and testing on the software was accomplished using this alternate accessible implement methods. Able-bodied friends also tested the games for me using traditional input.
Playing Tic-Tac-Toe

The Tic-Tac-Toe game can be found in the /TwoPlayerGame/examples folder. You should compile and upload the program to your two devices according to the procedures outlined on the previous page of this guide.

Player 1 always plays the "X" and Player 2 always plays the "O". It depends upon the state of the `IS_PLAYER_1` flag that you set when you compiled and uploaded the code to your device. The opening splash screen will tell you which one you have.

After the connection has been made between the two devices there will be a coin toss to decide who goes first. When it is your turn you will use the joystick to move your "X" or "O" around the 3 x 3 grid. It will appear green if the square is empty. It will turn red if you try to overwrite an existing symbol and it will not let you place your symbol there. When you have it in position, press the "Select" button. Your symbol will turn white. It will then be the other players turn. The image on the left shows the "O" getting ready to make the second move of the game. Note his symbol is green. On the right "X" tries to place in the same location and his symbol turns red indicating an illegal move.
Turns alternate until someone gets three in a row either horizontally, vertically, or diagonally or until all nine squares have been filled with no winner. In that case the game is declared a draw. Additionally to demonstrate the features of the system you can press the "Start" button to forfeit the game. When the game is over you will be prompted to press "Start" to start a new game. Make sure that you and your opponent don't start exactly simultaneously so that you are not both offering at the same time.

Playing Battleship

This program plays a classic game of "Battleship" where you place your five ships of varying sizes into your area of the ocean on a 10 x 10 grid. You then take turns with
your opponent firing shots at a grid coordinate in an attempt to sink all five of your opponents ships before they sink yours.

Using Sound Effects

This game has the option to add sound effects to your game. Using sound effects will take a little extra preparation before you upload your game software.

If you do not plan to use sound effects then you need to make a small change in the software before uploading. Open the Battleship.ino program using the Arduino IDE. On the second tab battleship.h at approximately line 22 you will see the following define statement.

```
#define USE_AUDIO true
```

Edit that line to read `false` and save and upload the program. Be sure to change the `IS_PLAYER_1` value on the front page before uploading to your second device. See the "Loading Sound Effects" section of the "Assembly and Setup ()" page above for instructions on how to load the sound effects files onto your device.

Compile and upload the game according to the procedures outlined in the "Demonstration Games ()" page setting the `IS_PLAYER_1` flag appropriately.

When you power on your device or press reset or upload the code initially you will see a splash screen followed by a menu that looks like the image below.

This will give you 3 options for placing your ships. The first option is the computer will randomly place all five of your ships on the board for you. The second option allows you to manually place the ships at the location and orientation that you want. The third option is for debugging and demonstration purposes. It places the ships at a fixed location and pre-sinks four of them so that you can fire just a couple of shots to end the game. Here is an animation showing how the random placement works. Notice it takes several tries of random locations to find a legal position.
Manual Ship Placement

If you choose the manual option on the opening menu, you will place each of your five ships yourself starting with the 5 square carrier, all the way down to the 2 square patrol boat. The ship will appear in a horizontal orientation at the first open square on the grid. You can press the "B" button and the ship will toggle back and forth between horizontal and vertical orientation. It will pivot around the small yellow dot which serves as your cursor. Use the joystick to position the ship where you want it and then press "Select" to place it. It will then call up the next ship until you have all 5 placed.

If you attempt to place a ship with part of it off the edge of the board or overlapping another previously placed ship it will not let you. The image below shows placement of the five slot Carrier in a horizontal or vertical position. Followed by an attempt to place the Battleship in the illegal position off the edge of the board.

NOTE: once you have placed a ship it cannot be moved without restarting the game.
Game Play

Only after you have placed your ships using either automatic or manual procedures will your device begin transmitting an offer of a game. If there is no response within a couple of seconds it will go into seeking mode seeking an invitation from the other device. As always, the offering player will press a button to initiate the coin toss and both devices will be informed of the results.

When it's your turn you will see a 10 x 10 grid with a green background. This is your "radar" screen. Using the joystick you move a yellow dot cursor around the screen and press "Select" to tell it where to fire at your enemy ships.

The game will not allow you to waste a turn by firing at the same location more than once. If your shot was a hit, the dot will turn red. If the shot missed, it will turn white.

Any time during your move you can press the "B" button to toggle the sound effects off or on. The image below shows the radar screen with the yellow cursor firing. The image on the right shows 2 misses in white and two red hits.

When it is your opponent's turn, your display will switch to a blue grid with your ships on it. This is the "sea" where your ships reside. When your opponent makes a move, if it hits one of your ships, that location will be marked with a red dot. If it's a miss, it will appear as a white dot in the sea. When all of the spots of a particular ship have been hit, the ship is sunk and it will turn completely red.
The five NeoPixels on your device indicate how many enemy ships you have sunk. Initially you have 5 green lights and they will turn red one by one when you sink an enemy ship.

When you hit a ship you will not know which ship you hit until you have completely destroyed it. It will then inform you what type of ship you sank. The images below show a submarine with 2 red hits and then after the third hit it turns completely red indicating it is sunk.

Play continues until one player loses all five ships. You can also press the "Start" button to resign your game. When the game is over you will be prompted to press "Start" to start a new game. The images below show a winning game and a game which has been forfeited as well as the restart message.

Creating Your Own Games

Overview

Our primary goal in this learning guide was to illustrate how to use the TwoPlayerGame Library to create your own two player games. The library code and example programs are extensively commented with detailed descriptions of each of the object classes and methods used in the game engine. In this section of the guide we will go over some of the basics of what you will need to do to create your own game.
We will not be covering the details of how to use the Adafruit_Arcada_Library to display graphics or to take input from the joystick and buttons. For details see this section of the PyGamer learning guide titled Arcada Library.

As mentioned earlier there are 3 example games provided. "Tic-Tac-Toe", "Battleship", and a debugging game consisting only of text input and output to the serial monitor. When developing your own game you might want to start by creating a copy of one of these 3 example games and editing out our game procedures and entering your own code in its place. We developed "Tic-Tac-Toe" by editing a copy of the demo and debugging game and in turn created "Battleship" by editing a copy of "Tic-Tac-Toe".

The library consists of 4 object class definitions. These are base classes with some abstract and virtual methods. You will have to create a derived class using this base class. The base classes handle all of the basic work of transmitting packets of data between the two devices and maintaining the general flow of the game. This means you can concentrate on creating the game specific portions of your code and let the game engine do the heavy lifting for you.

Here are the 4 base classes in the system:

- **baseRadio** -- An object that handles all radio communication between the devices.
- **baseMove** -- An object whose contents is transmitted to inform your opponent of your move.
- **baseResults** -- An object that returns information to you regarding the results of your move.
- **baseGame** -- The main object of the game engine. Handles basic game flow logic.

There is an additional object **basePacket** from which **baseMove** and **baseResults** are derived but you will not need to make your own extended version of it.

We have already designed an extension of **baseRadio** for the RF69HCW packet radio feather wing. If you are using that packet radio you can simply use our provided code. Eventually we hope to provide other options or perhaps one of you can design your own. We would welcome it as a commit to our GitHub repository.

In the following pages we will describe each of these classes and how you can extend them to implement your game.
Your Main Program

Let's take a quick look at our main program which is almost identical for all of the games. It will illustrate how the different parts of the game engine connect together to implement your game. Here is a complete listing of tic-tac-toe.ino.

```c
/*
 * Two Player Game Main Program
 * Compile and upload to one device using IS PLAYER 1 true and compile and upload
 * to the other device using IS PLAYER 1 false. This is the only difference between
 * the code on the two different machines.
 */
/*
 * A simple tic-tac-toe game.
 */
#define IS_PLAYER_1 true

//Basic game engine code
#include &lt;TwoPlayerGame.h&gt;

//Code for RF69HCW packet radios
#include &lt;TwoPlayerGame_RF69HCW.h&gt;

//All of my code is here
#include "tictactoe.h"

/*
 * The Game object requires pointers to Move, Results, and Radio objects. Create
 * an instance of each of these and pass their address to the game constructor.
 */
RF69Radio Radio;
TTT_Move Move;
TTT_Results Results;
TTT_Game Game(&amp;Move, &amp;Results, &amp;Radio, IS_PLAYER_1);

void setup() {
    Game.setup();       //Initializes everything
}
void loop() {
    Game.loopContents(); //Does everything
}
```

We have already discussed the use of the `IS_PLAYER_1` flag.

The basic game engine code is included via TwoPlayerGame.h.

The packet radio code for the RF69HCW is in TwoPlayerGame_RF69HCW.h. It defines an object type `RF69Radio`.

Finally we include the file containing our game code in this case tictactoe.h. This code defines 3 object classes: `TTT_Move`, `TTT_Results`, and `TTT_Game`. 
We then need to create an instance of the Radio, Move, and Results objects. We then pass their addresses to the constructor for the Game object along with the IS_P LAYER_1 flag.

From there you can see that the only item in our setup() function is a call to Game.setup(). Similarly the loop() consists of nothing but a call to the Game.loopContents() method.

The basePacket object class

The baseMove and baseResults classes are each derived from another class base Packet. While we do not need to create an extension of basePacket, Both of the baseMove and baseResults classes inherit its data and methods so we will examine it first. Data is transmitted in packets of binary data. The maximum size of a packet depends upon the type of radio you are using but for the RF69HCW it is 60 bytes per packet.

While we could have implemented data transfer in some more human readable format such as JSON, we risked bumping up against that 60 byte limit if a particular game needed to transfer a lot of data. By sending it in binary form we make it as compact as possible and we don't have to bother with encoding or decoding JSON even though there are libraries to assist us in doing so.

The code for the basePacket, baseMove, and baseResults objects are found in the files TwoPlayerGame_base_packet.h and TwoPlayerGame_base_packet.cpp.

Here is the definition of the basePacket class:

```
enum packetType_t {
    NO_PACKET_TYPE, OFFERING_GAME_PACKET, ACCEPTING_GAME_PACKET, MOVE_PACKET,
    RESULTS_PACKET, COIN_FLIP_PACKET
};
enum packetSubType_t {
    NO_SUBTYPE, NORMAL_MOVE, PASS_MOVE, QUIT_MOVE, NORMAL_RESULTS, HIT_RESULTS,
    MISS_RESULTS, WIN_RESULTS, LOSE_RESULTS, TIE_RESULTS, FLIP_TRUE, FLIP_FALSE
};
class basePacket {
public:
    baseRadio* Radio;
    packetType_t type;
    packetSubType_t subType;
    basePacket(void) {subType=NO_SUBTYPE;}
    basePacket(baseRadio* radio_ptr) {subType=NO_SUBTYPE; Radio=radio_ptr;}
    virtual size_t my_size() { return sizeof( *this ); }
    virtual bool send(void);
    virtual bool send(packetType_t t) {type=t; return send();}
    bool requireTypeTimeout(packetType_t t,uint16_t timeout);
    void requireType(packetType_t t);
#if(TPG_DEBUG)
    virtual void print(void); //Prints debug messages on the serial monitor.
```

©Adafruit Industries
Each packet has a pointer to the Radio object so that it knows how to send itself or be received into itself. It also has 2 other data items type and subType with the legal values defined in the enums shown above. These are the only data items contained in the basePacket and they are inherited by any classes based upon it. The methods are as follows:

- basePacket(void); and basePacket(baseRadio* radio_ptr); -- Constructors.
- virtual size_t my_size() { return sizeof( *this ); } -- Returns the size of the actual object as instantiated. We will explain later why this is necessary.
- virtual bool send(void); -- Sends the packet.
- virtual bool send(packetType_t t); -- Sends a simple non-data packet. Used in baseGame::offeringGame to send an OFFERING_GAME_PACKET and COIN_FLIP_PACKET. Also used in baseGame::acceptingGame to send an ACCEPTING_GAME_PACKET. Returns true if packet was acknowledged.
- bool requireTypeTimeout(packetType_t t,uint16_t timeout); -- Waits for the specified time in attempt to receive a particular type of packet from the other device. Returns true if the proper packet was received before timeout. Returns false if either time ran out or a received packet was the wrong type.
- void requireType(packetType_t t); -- Waits indefinitely for a packet of a particular type. Ignores any of other packets.
- virtual void print(void); -- Prints debug messages on the serial monitor.

Derived classes that have print() methods for debugging may want to call this function first. Note it is print() and not println().

How Packets Are Transmitted

We want to transmit data from a packet object on this device to an identical packet object on the other device. When sending data, the Radio object expects a pointer to the address where your data begins and a number that is the length of the data. Similarly when receiving data it expects and address to where the data should be placed and the maximum length of that location. Rather than copying the data into some buffer to be transmitted or to receive it into a buffer, we decided to just point to a location within the object itself and let the Radio object transmit directly out of the packet object on one device into a packet object on the other device. We had to be careful however that we didn't overwrite something that we didn't want to mess with.
If we start at address `this` (which is a pointer to the object itself) it would include the pointer to the function table. Although we are compiling identical code on identical devices we can't be 100% sure those addresses would be the same. In fact we have tested this code using PyGamer as one device and PyBadge as the other device. Similarly the first data item in a packet is a pointer to the radio object which we don't want to overwrite either. So we have to offset everything by `2*sizeof(void*)`. We know that `sizeof(void*)` is 4 but who knows... we might someday port this to a 64-bit platform Therefore using `sizeof(void*)` ensures we get it right and it self documents the code to explain where that 4 value came from.

WARNING: When implementing your derived Move and Results classes it is recommended you DO NOT make use of pointers to data unless you create a mechanism to send the pointed to data in a separate packet and then reassemble it on the other side.

In addition to needing to know where in the object we want to start transmitting data, we also need to know the size of the derived packet object. The virtual method `my_size()` correctly tells a base method the size of the actual object. Therefore the data we will transmit in each packet starts at `this+(2*sizeof(void))` and the length of the data is `my_size()-(2*sizeof(void))`.

NOTE: The `my_size()` gives the size of the object but object sizes are typically rounded off to multiples of 4. So if you have an odd amount of data, there will be some garbage data at the end to pad it out to a multiple of 4. Make sure you compile the code for both devices using the same compiler and the same settings to ensure that such padding occurs the same on both devices.

The baseMove object class

The `baseMove` class contains all the data for making and receiving moves. You MUST create a derived move class from it because it contains pure virtual functions that you will have to create. Your derived move class will contain whatever data and methods you need for your particular game. Here are details on the `baseMove`.

The code for the `baseMove` object is in the file TwoPlayerGame_base_packet.h and TwoPlayerGame_base_packet.cpp. The `baseMove` class is derived from the `basePacket` which is described in the previous section. The `baseMove` class inherits all of the data and methods of `basePacket`. Here is the definition of `baseMove`.

```cpp
class baseMove : public basePacket {
    public:
        uint16_t moveNum;
};
```
The following items are inherited from basePacket...

- **baseRadio* Radio;** -- The game engine initializes the Radio pointer automatically.
- **packetType_t type;** -- Is always MOVE_PACKET.
- **packetSubType_t subType;** -- Default is NORMAL_MOVE but can also be PASS_MOVE or QUIT_MOVE.

The following items are specific to a move packet

- **uint16_t moveNum;** -- The number of this move.
- **baseMove(void);** -- Constructor.
- **virtual size_t my_size() { return sizeof( *this ); };** -- Returns the size of the actual object as instantiated. See the discussion in the section on basePacket.
- **virtual void decideMyMove(void);** -- You MUST implement and override this pure virtual function. It will prompt the user for some input so he can specify his move using any method you want such as buttons, joysticks, or serial monitor input.
- **void require(void);** -- Wait forever for a MOVE_PACKET from the other device.
- **virtual void print(void);** -- Prints debug messages on the serial monitor.

**Creating a Derived Move Class**

Let's take a brief look at the TTT_Move class for the Tic-Tac-Toe game as well as the BShip_Move class for the Battleship game.

```cpp
class TTT_Move : public baseMove {
public:
    uint8_t square;
    size_t my_size() override { return sizeof( *this ); };
    void decideMyMove(void)override;
};

class BShip_Move : public baseMove {
public:
    uint8_t shot;
    size_t my_size() override { return sizeof( *this ); };
};
```
The only data we need to transmit for either of these games is the location that we selected during our move. In the Tic-Tac-Toe game we called it `square` and in the Battleship game we called it `shot` but it's just a single integer that is an index into the board denoting a location.

The real action goes on in the `decideMyMove()` method. We won't go through all the details of the `decideMyMove` implementations. You can look at the source code yourself which is well commented. The basic procedure is to place a cursor on the screen and then take input from the joystick using the Adafruit_Arcada_Library. When you make a selection, it first checks to make sure you didn't overwrite a previous selection. If it's in the clear then it copies the index into `square` or `shot` and then exits. You do not need to do anything to send your move. The game engine takes care of that for you. All you have to do is fill out the one data item and then exit.

The **baseResults object class**

The `baseResults` class defines a packet of information that is sent back to you after you make a move. Many games really don't require any returned information. For example in the Tic-Tac-Toe game, because the board is open you already know the results of your move. However in the Battleship game you need the other player to tell you if it was a hit or a miss. Even if you don't have hit and miss information that needs to be transmitted back, you still need to send a `Results` packet because it is the receiving player who determines if your move resulted in a win. So in the Tic-Tac-Toe game we have to send a results packet to acknowledge when the game is over.

You MUST create a derived move class from it because it contains pure virtual functions that you will have to create. Your derived move class will contain whatever data and methods you need for your particular game. Here are details on the `baseResults`.

The code for the `baseResults` object is in the file TwoPlayerGame_base_packet.h and TwoPlayerGame_base_packet.cpp. The `baseResults` class is derived from the `basePacket` which is described in the previous section. The `baseResults` class inherits all of the data and methods of `basePacket`. Here is the definition of `baseResults`.

```cpp
class baseResults : public basePacket {
    public:
        uint16_t resultsNum;
        baseResults(void) { type=RESULTS_PACKET; subType=NORMAL_RESULTS; }
        virtual size_t my_size() { return sizeof( *this ); }
};
```
virtual void require(void) {requireType(RESULTS_PACKET);};
virtual bool generateResults(baseMove* Move)=0;
virtual bool processResults(void)=0;
#if (TPG_DEBUG)
  virtual void print(void);//Prints debug information about the results. Calls basePacket::print
#endif
};

The following items are inherited from basePacket...

- baseRadio* Radio; -- The game engine initializes the Radio pointer automatically.
- packetType_t type; -- Is always RESULTS_PACKET.
- packetSubType_t subType; -- Default is NORMAL_RESULTS but can be any of the following: HIT_RESULTS, MISS_RESULTS, WIN_RESULTS, or LOSE_RESULTS.

The following items are specific to a results packet...

- uint16_t resultsNum; -- The move number to which these results refer.
- baseResults(void); -- Constructor.
- virtual size_t my_size() { return sizeof( *this ); }; -- Returns the size of the actual object as instantiated. See the discussion in the section on basePacket.
- virtual void require(void); -- Waits forever for a RESULTS_PACKET.
- virtual bool generateResults(baseMove* Move); -- You MUST implement a derived method in your game implementation to produce a packet of data to send back to your opponent telling them the results of their efforts. Here you will implement the rules that determine what is a WIN_RESULTS, LOSE_RESULTS, or TIE_RESULTS in addition to any other results information you may need to provide. Returns true if the results ended the game.
- virtual bool processResults(void); -- You MUST implement a derived method in your game implementation to react to the results packet you will receive after you make a move. Returns true if the results ended the game.
- virtual void print(void); -- Prints debug messages on the serial monitor.

Creating a Derived Results Class

Let's take a look at the TTT_Results and BShip_Results classes for our two demo games.

//Ways to win
enum win_t {NO_WIN, TOP_ROW, MIDDLE_ROW, BOTTOM_ROW, LEFT_COLUMN, MIDDLE_COLUMN, RIGHT_COLUMN,
class TTT_Results : public baseResults {
    public:
        win_t Win;
        size_t my_size() override { return sizeof( *this ); };
        bool processResults(void)override;
        bool generateResults(baseMove* Move)override;
    };

class BShip_Results : public baseResults {
    public:
        int8_t shipDestroyed;
        uint8_t shot;
        size_t my_size() override { return sizeof( *this ); };
        bool processResults(void)override;
        bool generateResults(baseMove* Move)override;
    };

For the Tic-Tac-Toe game we pass by a value that tells the type of winning move if Results.subType is WIN_RESULTS. We designed the game engine so that the results are the only determining factor of a win or lose game. That way we don't have to implement the rules in 2 different places.

For the Battleship game we have an signed integer that tells which if any ships were destroyed. If none were destroyed it is -1 and if one was destroyed it's 0 through 4. We also echo back to the player the location that he shot at. It saves us some extra work in the processResults() method.

We won't go through the details of the processResults() and generateResults(Move) implementations but you can look at the source code and see what we did.

The baseGame object class

The baseGame class defines a Game object that is the heart of the game engine. It controls the flow of the game through the opening negotiations to connect the radios, through the exchange of alternating moves, and then ultimately terminating the game and offering you the option to restart.

You MUST create a derived game class because it contains pure virtual functions that you must define. In our implementations we chose to make all of our game specific variables and functions globally accessible rather than make them data items or methods of the Game class.

The code for the baseGame object is in the file TwoPlayerGame_base_game.h and TwoPlayerGame_base_game.cpp.

Here is the definition of baseGame:
enum gameState_t {
    OFFERING_GAME, SEEKING_GAME, MY_TURN, OPPONENTS_TURN, GAME_OVER
};

class baseGame {
public:
    uint16_t currentMoveNum;
    baseMove* Move;
    baseResults* Results;
    baseRadio* Radio;
    uint8_t myPlayerNum; //For initializing my radio
    uint8_t otherPlayerNum; //Destination of our transmissions
    baseGame(baseMove* move_ptr, baseResults* results_ptr,
              baseRadio* radio_ptr, bool isPlayer_1);
    virtual void setup(void);
    virtual void loopContents(void);
protected:
    virtual void initialize(void){gameState=OFFERING_GAME;};
    virtual bool coinFlip(void)=0;
    virtual void processGameOver(void)=0;
    virtual void fatalError(const char* s)=0;
    virtual void processFlip(bool coin) {};
    virtual void foundGame(void) {};
    gameState_t gameState; //The internal state of the game engine
private:
    //Internal routines that handle each of the various states of the engine
    void offeringGame(void);
    void seekingGame(void);
    void doMyTurn(void);
    void doOpponentsTurn(void);
    void gameOver(void);
};

Here are the details of the baseGame class...

- uint16_t currentMoveNum; -- The number of the current move.
- baseMove* Move; -- Pointer to a move object that will be transmitted between devices. Is used for both sending our move and receiving our opponents move.
- baseResults* Results; -- Pointer to a results object that will be transmitted between devices. Is used for both sending our results and receiving results from our opponent.
- baseRadio* Radio; -- Pointer to a radio object that will do the transmitting and receiving of data.
- uint8_t myPlayerNum; -- My player number either 1 or 2 computed by constructor based on isPlayer_1 parameter.
- baseGame(baseMove* move_ptr, baseResults* results_ptr, baseRadio* radio_ptr, bool isPlayer_1); -- Constructor.
- virtual void setup(void); -- Call this method ONLY once in your main setup() function.
- virtual void loopContents(void); -- Call this method inside your main loop() function.
- virtual void initialize(void); -- This method is called any time a new game starts.
• **bool coinFlip(void);** -- You MUST implement this pure virtual method in your derived game class. It determines who goes first.
• **void processGameOver(void);** -- You MUST implement this method in your derived game class. It will be called at the end of every game. You get to decide what to do.
• **void fatalError (const char* s);** -- You MUST implement this method in your derived game class. It would only be called in the event of an unrecoverable error such as failure to receive acknowledgment of a data packet or some other programming logic error.
• **void processFlip(bool coin){};** -- Used by accepting player to process an incoming `COIN_FLIP_PACKET`. The base method does nothing.
• **void foundGame(void) {}**; -- Used by accepting player to print a message that a game has been found and we are waiting on the offering player to do the coin toss.
• **gameState_t gameState;** -- The internal state of the game engine. See the `gameState_t` definition above for legal values.

The remaining methods are all internal methods for processing the various game states.

We suggest you take a look at the implementation of each of these methods in the Tic-Tac-Toe and Battleship games. A couple of extra notes to consider.

When you are the accepting player and you are waiting on the other player to perform the coin toss, you don't know that your game has been accepted. So the offering player will send a `FOUND_GAME_PACKET` to let you know that you have found a game. When you get that packet, the game engine calls `foundGame()` so that you can display a message saying that you have found a game and are waiting on the coin toss. Similarly if you lose the coin toss, you have to wait for your opponent to make their first move. Therefore the results of the coin toss are sent to you in a `COIN_FLIP_PACKET` and the game engine calls `processFlip(coin)` so that you can print a message explaining that you lost the toss.

In both demonstration games we implemented `coinFlip()` as a random coin toss but you could implement any system you want as long as it returns true for the offering player winning and false for the accepting player winning. For example you might implement a rock/paper/scissors decision.

Note that your `setup()` and `initialize()` derived methods MUST call the base methods somewhere in their processing. See the example games for details.
We invite you to take a close look at the implementation of `loopContents()`. The base method code is shown below.

```c
void baseGame::loopContents(void) {
    switch(gameState) {
    case OFFERING_GAME:  offeringGame();  break;
    case SEEKING_GAME:   seekingGame();   break;
    case MY_TURN:        doMyTurn();      break;
    case OPPONENTS_TURN: doOpponentsTurn(); break;
    case GAME_OVER:      gameOver();      break;
    }
}
```

As you can see it simply hands off control to the proper routine depending on the `gameState`. But under some circumstances you might want to do something before or after the game engine enters a particular state. Take a look at the `BShip::loopContents()` method we implemented. With some of the details removed, it looks like this.

```c
void BShip_Game::loopContents(void) {
    gameState_t saveState=gameState;  //save the gameState for postprocessing
    //This is processing we do before the game engine does a particular state
    switch(gameState) {
    case OFFERING_GAME:
        //Do some stuff before we enter the offering state.
        break;
    case SEEKING_GAME:
        //Do some stuff before we enter the streaking state.
        break;
    case MY_TURN:
        //Do some stuff before each of my turns.
        break;
    case OPPONENTS_TURN:
        //Do some stuff before each of my opponent's turns.
        break;
    }
    //You MUST call this to let the game engine do its thing
    baseGame::loopContents();
    
    //This is processing we do after the game engine does a particular state
    switch(saveState) {
    case OFFERING_GAME:
        //Do some stuff after offering of the game.
        break;
    case OPPONENTS_TURN:
        //Do some stuff after the opponents turn.
        break;
    }
}
```

Note that we had to save the state of the game because after we call `baseGame::loopContents()` the variable `gameState` is no longer the same. The `switch` statement after we call `baseGame::loopContents()` uses `saveState` and not `gameState`. 
You might not need to implement all of the options before or after your \texttt{loopContents()} method. The Tic-Tac-Toe game doesn't do any after-loop processing and neither example uses the \texttt{MY_TURN} option for pre-processing or post-processing. We just wanted to show you that you have the option to do this if necessary.

### The baseRadio object class

As previously mentioned, we have already created a radio object class for the RFM69HCW radios. You can use it in this project and not worry about creating your own. However if you would like to implement some other means of transmitting data between your devices you would start with the \texttt{baseRadio} class and make an extension of it with the specifics for your particular radio.

Our implementation of the \texttt{RF69Radio} class is based upon the \texttt{RadioHead} Library, therefore our \texttt{baseRadio} class closely resembles some of the methods used in \texttt{RadioHead}. If you create another transport system that is not based on \texttt{RadioHead} you will have to conform to our API which might be a little more difficult.

\begin{quote}
\textbf{NOTE:} The RadioHead library refers to our device as an "RF69". Throughout our code we refer to it by that name or perhaps "RF69HCW". However the actual name of the Radio FeatherWing is "RFM69HCW" with an "M" in it. We didn't notice this until the code was written. We chose not to rewrite everything with the extra "M". A search and replace would have messed up the RadioHead references which needed to remain "RF69".
\end{quote}

\begin{verbatim}
class baseRadio {
    public:
        uint8_t myPlayerNum;    //my radios address
        uint8_t otherPlayerNum; //opponent's radio address
        virtual bool setup(uint8_t myPlayerNum, uint8_t otherPlayerNum)=0;
        virtual bool send(uint8_t* packet_ptr, uint8_t len)=0;
        virtual bool recvTimeout(uint8_t* packet_ptr, uint8_t* len_ptr, uint16_t timeout)=0;
        virtual bool recv(uint8_t* packet_ptr, uint8_t* len_ptr)=0;
        virtual bool available(void)=0;
    };
\end{verbatim}

Note that this is an abstract class that cannot be instantiated itself. All of the methods are pure virtual so you will have to implement all of them.
Each device has its own unique address. Player 1 will be device #1 and Player 2 will be device #2. This device number is the ONLY difference between the software on one device versus the other.

- **bool setup(uint8_t myPlayerNum, uint8_t otherPlayerNum);** -- Gets called ONCE during your Game.setup() call inside your main program setup(). Returns true if the setup was successful.
- **bool send(uint8_t* packet_ptr,uint8_t len);** -- Sends a packet of data starting at memory address packet_ptr with the number of bytes specified by len. Returns true if sent data was acknowledged as received.
- **bool recvTimeout(uint8_t* packet_ptr,uint8_t* len_ptr,uint16_t timeout);** -- Attempts to receive a packet and waits until the specified timeout. The packet_ptr is the start address of the data. The len_ptr points to a uint8_t specifying the size of our buffer. Upon return it passes back the actual number of bytes received. Returns true if data was received before the timeout.
- **bool recv(uint8_t* packet_ptr,uint8_t* len_ptr);** -- If data is available it receives it. Parameters are the same as the first two of recvTimeout explained above. Returns true if successful.
- **bool available(void);** -- Returns true if data is available to be received.

See the discussion on basePacket about how we determine the start address in the length of the data to send.

**Our RF69Radio object class**

The RadioHead library offers an advanced packet transmission system called RHReliableDatagram that includes encrypting the data, an automatic acknowledgment of a packet received, and multiple retries if an initial transmission doesn't get through. All of this is completely transparent to us. We just send or receive using the methods we defined in our baseRadio class which in turn call similar methods in the RHReliableDatagram library.

Every packet which is sent is acknowledged by the receiving device using a system that is invisible to us. So for example if send a Move packet, when it is received by the other device it sends a hidden acknowledgment back to us to let us know the move was received. This is not the Results packet that we ourselves send back. This is a behind-the-scenes acknowledgment. When the other device sends a Results packet in response to our move, it also is acknowledged. Only after multiple failures to receive an acknowledgment do we get a packet error. This makes for extremely reliable data transmission. If you are implementing a different type of radio
and it is supported by RadioHead and RHReliableDatagram then you should definitely take advantage of that system.

The only errors we are likely to see is if one device is too busy perhaps playing a sound file or doing something else that would prevent it from sending an acknowledgment signal. This would cause a fatal error in our code. At some point we might try to implement a way to recover such mistakes but keep in mind that we only get this failure after the RHReliableDatagram system has already tried multiple times so it is unlikely we can do anything to recover from such errors. In implementing the sound effects in the Battleship game we created a special myDelay(amount) function to use instead of the traditional Arduino delay(amount). It includes a yield() statement and it seemed to solve many of our conflicts between packet transmission and sound effects.

Other Support Software

In addition to the software already described, the TwoPlayerGame Library also contains two other useful files.

The accessible input feature we described earlier that allows you to stimulate joystick and button presses using the serial monitor is available in TwoPlayerGame/AccessibleArcada.h. To invoke it we have included the following code in all of our sample games.

```c
//Set this to true for an alternate assistive technology input using serial monitor
//input in addition to the joystick and buttons.
#define ACCESSIBLE_INPUT false

#if(ACCESSIBLE_INPUT)
    #include &lt;AccessibleArcada.h&gt;   //alternate input system for assistive technology
    AccessibleArcada Device;
#else
    Adafruit_Arcada Device;
#endif
```

Another useful module is TwoPlayerGame_wave.h which contains the setup and playback code for the sound effects in the Battleship game. This could be used in other games that require sound effects. The Adafruit_Arcada_Library has wave playback methods called:

```c
wavStatus WavPlayComplete(char *filename);
wavStatus WavPlayComplete(File f);
```

These methods make it easy to use an internal player to playback wave files however our experience was that the sound quality was not as good as using a custom player.
It could be interference from the radio handling software or some problem with the built-in player not handling our particular variety of wave files properly. We didn't bother to track it down. We like the sound quality we get with our custom player so we left it in. It can be used to playback sound files in other games.