# VDM-22 Sapfir

**https://youtu.be/B7cTL-qhFzI**

# Design Overview

### What is the VDM-22?

The VDM-22 (Vector Drawing Machine – 22) is a rudimentary replica of the famous RP-21/RP-22 Sapfir radar’s pilot interface. The defining feature of the VDM-22 is its graphics system – an 8-bit dual digital-to-analog converter generating vector graphics for display on an analog oscilloscope operating in X-Y mode.

The VDM-22 is capable of displaying both the B-Scope (before lock-on) and the C-scope (the chase mode). When the pilot starts the simulator, VDM-22 will draw a simulated bandit at the pilot’s 12 o’clock. The pilot will then have to maneuver his/her plane to achieve a lock on the bandit.

Once a good lock is achieved, the VDM-22 will proceed to the C-scope, like its real-life counterpart. The C-scope is a forward-looking scope; its interfaced was designed so the pilot can fly the plane toward the target in 0-visibility conditions. In this scope, the pilot will need to again maneuver the aircraft to match the bandit’s evasive maneuvers, while keeping an eye on the RWR (radar warning receiver) to deploy countermeasures when the bandit attempts to lock his/her aircraft.

When the bogey approaches around 2/3 of the ranging scale (the horizontal line on the C-Scope), the pilot will have to press the launch button to win the game. Launch authority toggle switches will have to be turned on before the launch button is pressed for VDM-22 to register the launch action.

There are three ways to fail the simulation. The first occurs when the pilot fails to keep the bandit within the radarscope using his/her joystick. The second happens when the pilot fails to launch the missile before the bandit approaches within 1/3 of the distance scale in C-scope. The third is when the pilot fails to react in time (5 seconds) by releasing countermeasures when the RWR prompts for an enemy lock.

A picture containing text, device, meter, different

Description automatically generated

### Design Evolution

The original design idea of the VDM-22 is to implement the F-14 tomcat’s powerful AN/AWG9 radar operated entirely by the RIO (Radar Intercept Officer). Like the MiG-21’s RP-22, the AN/AWG9 also has a CRT-based vector display, albeit much bigger in size. The difficulties of implementing such a design using the ATMega328 microcontroller soon became obvious:

1. The F14’s B-scope and C-scope utilize two separate CRT displays, which means only one operating mode will be available at once if implemented in the project.
2. The F14’s vector display has a much more complex user interface, with multiple characters, diagonal lines, circles, and numbers. The ATMega328’s limited computational capabilities and program space made it impossible to load all the graphics maps in memory while keeping the program running at a reasonable speed.
3. The controls of the AWG9 radar, while rather comprehensive and easy to use (in 70s standards) still require at least 25-30 switches to operate at normal capacity. The space constraint of having to mount the entire machine in the Tektronix 5110 mainframe made it impossible to implement so many switches on board. And the idea of a 30-switch external controller is a bit too complicated for any player (and our programmer) to understand within our time constraints.
4. During initial testing, we were able to determine that the resolution of the 5D10 digital waveform/storage plug-in (8bit @ 1MHz) used in my Tektronix 5110 oscilloscope is incapable of displaying the fine details of the AWG9 radar interface. This discovery delivered the last straw for the team’s decision to implement the relatively simple RP-22 interface instead.

Another significant design change is in the controls quadrant. The original design idea is to have the MiG-21’s symbolic left-curving center control stick 3D-printed and mounted on a “decapitated” Sparkfun joystick module. It later turned out to be infeasible as the properties of 3D-printed plastic made it extremely difficult to route any wire for the buttons through. The final design opted for a normal control quadrant with the switches on top and the original Sparkfun joystick mounted below.

## Design Verification

**Enclosure Verification**

Laser cutting models are directly loaded in the software, and we are able to see what it looks like before it gets printed. As our enclosure design is very simple, no extra steps rather than direct inspection is needed.

**Hardware Verification**

Due to the complex nature of the design, no SPICE verification was conducted. The design was instead tested out on a breadboard connected to an Arduino UNO clone. The corresponding pins of the Arduino UNO were connected to the DAC situated on the breadboard. Despite the high switching speed, we managed to obtain a decent display on a testing BKPrecision oscilloscope which we shown in class during the project update presentation.

**Software Verification**

Because of the design complexity, it is impossible to debug simply using serial print from Arduino. The QT Framework is used to aid the debugging process, as it comes with a great GDB tool and a UI Framework.

### My Quick Painter [ MyQuickPaintedItem.h extends QQuickPaintedItem.h]

An Image buffer is encapsulated in this *MyQuickPaintedItem* for the *QPainter* to pull from synchronously before it knows what to draw on QML Canvas. This *QImage* object is 256\*256 in dimensions, which matches our oscilloscope. *QImage::Format\_Mono* format is used as our oscilloscope does not support color display.

QImage imageToBePrinted;

The *paint()* function is reimplemented from *QQuickPaintedItem***,** as it provides an API to use the legacy *QPainter* to paint an *QImage* to a QML Canvas using OpenGL. The second function clearScreen()simply clears up the image buffer imageToBePrinted to a blank screen. Lastly, displayPixelOnScreen()will update the pixel information to the image buffer.

void *paint*(QPainter \*);

bool clearScreen();

bool displayPixelOnScreen();

### Communicator Object [ QtCommunicator.h extends Communicator.h ]

The basic functionalities of Communicator Object interface are explained with details in **Software Design Implementation** section. Here the Communicator Object is reimplemented in a QT way, different from *unocommunicator.h* which reads input and writes output from and to digital pins, that all inputs will be read from QML buttons and sliders and outputs will be written to QML Rectangles. Apart from the overridden functions, several new are introduced.

For outputs, three pointers rectangleRwr1Led, rectangleRwr2Led, and rectangleBuzzer pointing to QML Rectangles, the three LEDs below the circular screen (stored as its base type QQuickItem\*) are encapsulated so that QML Rectangles can be called in C++. In the same way, a pointer of our customized painter myQuickPaintedItemPointer is encapsulated as well. Similarly for inputs, buttonIntermediateBuffer contains the current state of four buttons and sliderIntermediateBuffer contains the current position of two buttons, both is updated real-time in the UI thread.

QPointer<MyQuickPaintedItem> myQuickPaintedItemPointer;

QPointer<QQuickItem> rectangleRwr1Led;

QPointer<QQuickItem> rectangleRwr2Led;

QPointer<QQuickItem> rectangleBuzzer;

QVector<bool> buttonIntermediateBuffer;

QVector<qreal> sliderIntermediateBuffer;

### My Game Server (extending QObject) [ MyGameServer.h ]

The MyGameServerObject matches the top level .ino file, which works similarly to the setup and loop callbacks in Arduino IDE. The reason why an extra layer being necessary in QT is because direct access to the main event loop is impossible, so our customized event loop must be defined.

The *timerEvent()* works the same with the Arduino loop callback. This function will be called indefinitely by a set period, which is defined in the constructor of MyGameServer Object.

void *timerEvent*(QTimerEvent \*);

The Playable Object is instantiated and encapsulated in the MyGameServer Object. This step is done differently in Arduino, as an Playable Object should be manually instantiated globally. However, MyGameServer Object in this case acts like a bigger Playable Object, which will be manually instantiated in main.cpp from QT after all QML components are registered.

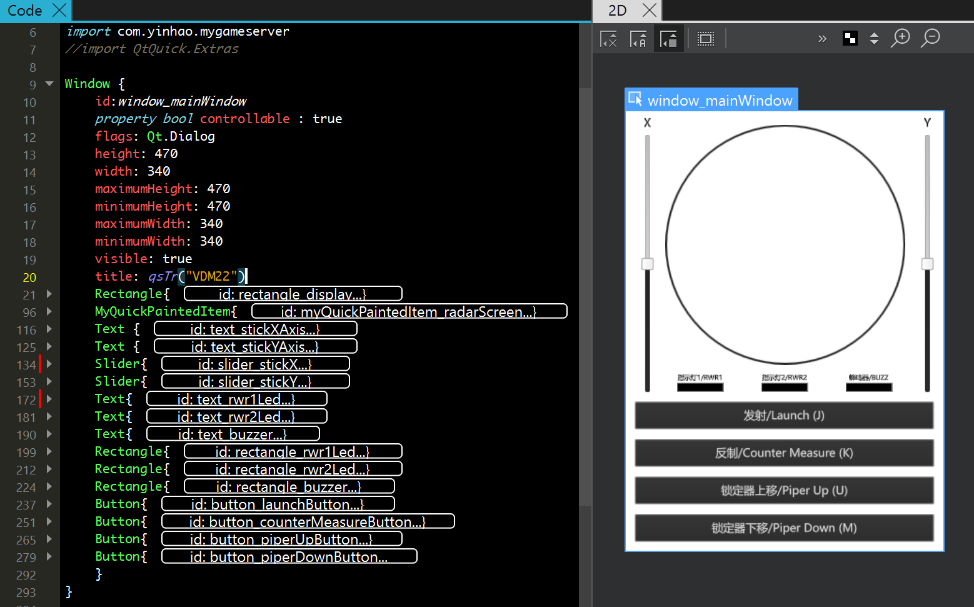
Playable<QT\_TEMPALTE\_PARAM> playable;

Some slots are defined so that when any button or slider is toggled, a signal will be sent to the event handler, and the corresponding slot function will be called. Q\_INVOKABLE macro is added before these slots so that these functions will be registered in the Meta-Object System as .moc file and subsequently can be directly in QML as a JavaScript function.

Q\_INVOKABLE bool onButtonValueChanged(QString, bool);

Q\_INVOKABLE bool onSliderValueChanged(QString, qreal);

The last step is to simulate the user input output system and the oscilloscope display, so a quick UI interface skeleton is implemented in QML:



Some trivial lines of code needes to be added in the main.cpp. Our customized QObject of *MyGameServer* and *MyQuickPaintedItem* are registered as a QML Type:

qmlRegisterType<MyQuickPaintedItem>("…", 1, 0, "MyGameServer");

qmlRegisterType<MyQuickPaintedItem>("…", 1, 0, "MyQuickPaintedItem");

At the same time, create instances of MyGameServer Object and reimplemented Communicator Object:

MyGameServer myGameServer{};

QtCommunicator qtCommunicator{};

Use the QQmlContext engine to search for the pointers pointing to the three QML Rectangles representing three LEDs, and the MyQuickPaintedItem Object, aka my painter, from the Meta-Object tree, and passing them to the reimplemented Communicator Object:

QQmlContext\* a = engine.rootContext();

a->setContextProperty("myGameServer", &*myGameServer*);

MyQuickPaintedItem\* myQuickPaintedItem = qobject\_cast<MyQuickPaintedItem\*>(engine.rootObjects().at(0)->findChild<QQuickPaintedItem\*>("…"));

QQuickItem\* rectangleRwr1Led = qobject\_cast<QQuickItem\*>(engine.rootObjects().at(0)->findChild<QQuickItem\*>("…"));

QQuickItem\* rectangleRwr2Led = qobject\_cast<QQuickItem\*>(engine.rootObjects().at(0)->findChild<QQuickItem\*>("…"));

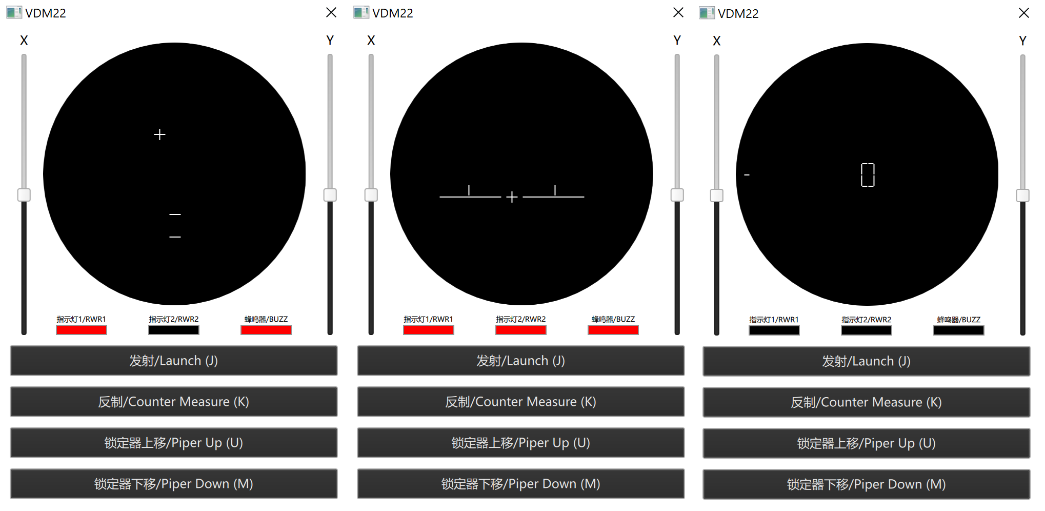
QQuickItem\* rectangleBuzzer = qobject\_cast<QQuickItem\*>(engine.rootObjects().at(0)->findChild<QQuickItem\*>("…"));

qtCommunicator.setMyQuickPaintedItem(*myQuickPaintedItem*);

qtCommunicator.setQQuickItemLeds(*rectangleRwr1Led*, *rectangleRwr2Led*, *rectangleBuzzer*);

myGameServer.setCommunicator(&*qtCommunicator*);

After doing all that, a quick visualization and operation of a game play is available on the PC:



When all the game behaviors fall within our expectation, the software implementation is deemed to be verified, and ready to be ported to the hardware.

**Electronic Design Implementation**

### Schematic Design[[1]](#footnote-1)

The entire design is broken down into three separate schematics – the Processor Board, the Joystick Board, and the RWR board, which will be explained separately.

1. Processor board

The processor board contains all the major logic; power supply; the DAC and both digital and analog I/O sections.

Power to the board is provided by a 10-35V DC supply. Both a battery interface for two parallel A23 12V batteries and a barrel jack for any standard DC supply is included. The supply voltage is then regulated to 5V by an LM7805 voltage regulator. There are two major reasons for this design choice. **The first** is a planned revision 2 that takes power directly from the bus architecture of the Tektronix 5000 series mainframe oscilloscope. The Tektronix bus is capable of supplying 12V and –12V DC. With a properly designed conversion board, the oscilloscope itself can directly power the VDM-22. **The second** reason is the reduction in both radiating and ripple noise generated from the incoming AC/DC supply by using the well-designed regulator system. While the LM7805 is still fundamentally a switching converter, it is indeed much quieter compared to the Hewlett Packard 18.5V power supply used during testing.

A low-power rail/ground rail noise is extremely critical in the design process of VDM-22. Being a mixed digital/analog design, the fast-switching element of the digital circuitry may cause significant noise to the analog power/ground rail of the DAC. With the output device being an oscilloscope, even the slightest noise will alter the display dramatically. The power rail is comprehensively decoupled with special attention paid to the DAC. Separate analog / digital ground nets are also included, with the two connected using a 0-Ohm resistor/jumper for minimum noise leakage.

The digital I/O section consists mainly of headers for connection to the RWR board (doubles as serial debug output), launch authority switch quadrant, and the GND net. The entire system bus is also available (for debugging and system extension purposes) on an 18-pin header, named PDS (Processor Direct Slot) after Apple’s 68020/030-expansion slot. Off-board connections (for both joystick and buttons) are provided in the form of 2 “USB” connectors. While not actually implementing the USB architecture at all, the single-orientation plug combined with its structural ruggedness, availability of cables, and ease of use made it the ideal connector for off-board connections to a controller.

The analog I/O section consists of two BNC connectors for direct interfacing with BNC cables. BNC coax cables are known for their excellent noise resistance and have been the standard for oscilloscope probes since the mid-1970s. They are grounded through the analog GND plane and impedance matched for optimal performance.

The heart of the VDM-22 is the perfect combination of the ATMega328 microcontroller and the Analog Devices AD7302 dual 8-bit DAC. The AD7302 is chosen for its single-rail operation capabilities and low power consumption (only 15mW), which simplified our power supply by a whole magnitude. Its lenient input timing requirement and fast parallel data interface meant it is possible to control it with a relatively slow microcontroller like the ATMega328. The AD7302’s rail-to-rail swing capability coupled with the fast-settling time of only 1.2 us also made it ideal for outputting to the oscilloscope.

1. Controller Support Board

The controller board consists of the Sparkfun joystick module, two USB2.0 Type A receptacles for connections to the processor board and 4 2-pin headers for the buttons. The original design idea, as mentioned before in part I.2, is to attach an actual flight stick to the top of the joystick module. Thus the built-in push button on the joystick is not connected in favor of standalone lock buttons. This feature is retained in the final revision.

1. RWR Board

The radar warning receiver is a very straightforward design. 4 LEDs arranged in a common cathode configuration with one 3-pin connector for connection to the processor board.

### PCB Design

The RWR and Joystick support boards utilize standard design placement and auto generated nets connections, so this document will focus on the processor board design.

The processor board features 90% manually wired traces with separate digital and analog ground planes, to reduce noise in the power, ground and analog transmission nets. The PCB can be broken down into four sections: power regulation, digital logic, analog processing and speaker/off-board connectors.

1. Power regulation

Two connectors are provided for power input, a standard DC barrel jack for regulated DC and a snap-on terminal block connector for battery connection. Following a new design principle I discovered when looking through the many handy IBM technical references, a “fools diode” is placed in parallel between the input DC Vcc rail and GND rail. When the power supply (or the battery) is plugged in with the incorrect polarity, the 1N4001 diode will enter the forward-biased mode, shorting out the power supply thus protecting the sensitive ICs from reverse polarity. After all over-current protection is more the job of the power supply than the PCB, and the A23 battery comes with a current limiter on its own.

The unregulated DC then enters the LM7805 voltage regulator for stepping down to 5V. Due to my inexperience with power systems outside recapping them, a heat sink slot was not included for the LM7805. During the testing phase, it was discovered that input voltages of +15V would immediately raise its temperature to over 100 degrees Celsius. To reduce the risk of fire, the internal enclosure was then changed from wood to acrylic with a Raspberry Pi memory controller heat sink glued onto the regulator. A slot for a screw-in heat sink will be added in future revisions.

1. Digital Logic

The digital logic section is kept close to the power regulation for a faster return path to GND. The logic traces are also kept as compact as possible to minimize noise leakage to nearby analog traces. Due to the properties of DIP, it is difficult to keep all the digital traces on one side of the board. So, the PDS (Processor Direct Slot) acted as the vias for logic pins on the far side of the microcontroller.

A programming block is included for uploading using the ATM programmer. It is also capable of supplying unfiltered +5V power to the entire board, although a lot noisier compared to power coming from the regulator circuitry.

1. Analog Processing

The DAC sits directly across the digital and analog ground planes. Thanks to Analog Device’s excellent DIP layout, the majority of the digital pins are kept on the left side with only the slower-switching DAC select and DAC load positioned on the right side. This enabled me to use the DAC as a “bridge” between the digital and analog section. A section of the analog GND plane is cut out to avoid EM issues when a trace crosses two different ground planes.

The DAC is directly connected to the BNC connectors for analog output. Line impedance is provided in the form of two aerospace grade 1% 100-ohm resistors, with optional high-frequency decoupling capacitors available.

The analog ground plane then goes all the way around the entire digital processing region, the off-board IO region to rejoin with the power connectors using a 0-Ohm 1% jumper. This design was proven during testing to have eliminated the majority of visible noise, when compared to the breadboard version displayed during initial presentation.

1. Speaker/Off board connectors

The output connectors (2 USB 2.0 receptacles) are situated on the bottom right of the PCB. Pull-up resistors and load resistors are next to the I/O. The Piezo beeper is situated above the load resistors for a cleaner look.

**Software Design Implementation**

The final design is a null-safe, cross-platform library that can be used on virtually any platform.

The process to include the game in any platform is extremely simple:

1. Implement Communicator Object interface by reimplementing **only** the virtual functions.
2. Include the implemented communicator header file. (*unocommunicator.h* in our case) and the *playable.h* header.
3. Create an instance of the implemented communicator and Playable object, and pass the pointer to playable by calling setCommunicator().
4. Call the event loop executeCycle() indefinitely.

For instance, our top-level sketch file running on Arduino IDE will look like this:

#include "playable.h"

#include "unocommunicator.h"

Playable<UNO\_TEMPALTE\_PARAM> playable;

UnoCommunicator unoCommunicator(*false*);

void setup() {

playable.setCommunicator(&unoCommunicator);

}

void loop() {

playable.executeCycle();

}

How is this library developed? For the complexity of this design schematics, OOD can significantly make the code more readable. There are three Objects in our final release: the communicator, the displayable, and the playable. Templates are extensively used so the code can be reused independent of platforms:

*template* <unsigned char B, *typename* T, *typename* D, *typename* A>

The following implementation is used for the design:

|  |  |  |
| --- | --- | --- |
| **Template** | **Meaning** | **Our Implementation** |
| B | Number of bits of digital signals | 8 |
| T | Data type that coordinates will be stored as | 16-bits signed integer |
| D | Data type that digital signals will be stored as | 16-bits signed integer |
| A | Data type that analog signals will be stored as | Floating-point number |

*\*All implementations are included inline in .h files. No .cpp files are needed.*

### 1. Communicator Object [ UnoCommunicator.h extends Communicator.h ]

The Communicator object acts like a bridge between the input/output pins and the Playable object. Data read from the GPIO pins would be stored in the following buffer before they are used by the Playable by calling the getter function *platformSpecificUpdatePinsToBuffer*():

bool digitalInputLaunchButtonBuffer;

bool digitalInputCounterMeasureBuffer;

bool digitalInputPiperUpButtonBuffer;

bool digitalInputPiperDownButtonBuffer;

A analogInputStickXAxisBuffer;

A analogInputStickYAxisBuffer;

Similarly, data to be written to GPIO pins would need to be stored to the following writing buffer before they are updated by the setter function *platformSpecificUpdatePinsToBuffer*():

bool digitalOutputBuzzerBuffer;

bool digitalOutputRwr1LedBuffer;

bool digitalOutputRwr2LedBuffer;

D analogOutputVectorGraphicsXAxisVoltageBuffer;

D analogOutputStickYAxisSliderVoltageBuffer;

Including the setter and getter functions mentioned above, all functions in the Communicator object are purely virtual and should to be overridden in each different platform. Additionally, *platformSpecificFlashPixelToScreen()* will flash all pixels to the screen (oscilloscope in our case) **exactly once**, and *platformSpecificRandomGenerator()* generates a random number within a given range, which is used to simulate random movements of the enemy plane in our design.

*virtual bool platformSpecificUpdateBufferToPins();*

*virtual bool platformSpecificUpdatePinsToBuffer();*

*virtual bool platformSpecificFlashPixelToScreen(const T, const T);*

*virtual T platformSpecificRandomGenerator(const T, const T);*

During runtime, a platform-specifically implemented instance of Communicator Object (***Unocommunicator*** from *unocommunicator.h,* which provides the ATMega328 implementation of the Communicator interface, is created globally on the heap while a copy of its pointer, dynamically casted to the base form ***Communicator*\***, is stored in the Playable object.

### 2. Displayable Object [ Displayable.h ]

The Displayable object provides a way to pre-allocate memory space in the heap for all the pixel information of each object displayable on the screen. Each movable entities displayable on the screen must maps to a distinct instance of Displayable object.

A Pixel is a simple struct that contains a pair of T values, representing the relative x and y coordinates. Each Displayable Object has a fixed-sized array of Pixel that stores positions all the pixels making up the displayable entity:

Pixel pixelArray[P];

The visibility property that tells the screen flashing function from the Communicator Object *platformSpecificFlashPixelToScreen()*whether or not to actually flash this entity to the screen.

bool visibility;

Lastly, *initializePixelInformation()*is called whenever a re-positioning of this displayable entity is needed, and its argument of 4*\*G* two-dimensional array containing the information of how many line segments are there as G, x-starting position as[G][0], x-ending position (inclusive) as [G][1], y-starting position as [G][2], x-ending position (inclusive) as[G][3] . pixelArray is **read-only**, and any attempts to modify the pixel information without explicitly calling *initializePixelInformation()* could potentially result in unexpected displaying behaviors:

bool *initializePixelInformation*(*const* T (&)[G][4]);

To avoid memory segmentations, the number of pixels P of each displayable entities must be calculated during **compile time** before it can be instantiated on the heap:

*template* <unsigned int P, *typename* T>

### 3. Playable Object [ Playable.h ]

The Playable Object is the main framework object that defines how the game should behave. Some parameters can be customized by directly changing the macros defined in *Playable.h*:

|  |  |
| --- | --- |
| **Macro Name** | **Definition** |
| LINE\_WIDTH | Number of pixels of line width |
| CR\_HEIGHT | C-Scope displayable reference height |
| CR\_WIDTH | C-Scope displayable reference width |
| BR\_HEIGHT | B-Scope displayable reference height |
| BR\_WIDTH | B-Scope displayable reference width |
| PLANE\_WIDTH | Plane displayable width |
| DIGIT\_WIDTH | 7-Segment displayable reference width |
| MAXABS\_COORDINATE | Maximum absolute coordinates of screen border |
| DIGIT\_DISPLAYTIME | Duration of display for scores and “END” |
| PIPER\_FREQ | Speed of movement for piper |
| REFERENCE\_FREQ | Speed of movement for B-Scope reference |
| TIMER\_PERIOD | Clock period |
| AUTOMOVE\_FREQ | Speed of random movement |
| DISTANCE\_FREQ | Speed of B-Scope closing |
| BLINK\_FREQ | Frequency of LED blinks |
| ATTACK\_PROBABLITY | Frequency of counter-measure |

The communicatorPointer stores the pointer of an instantiated Communicator object. However, the program is a null-safe, which means the game can still run without a valid communicator, but there is no way to control the game and to display the displayable entities. No exceptions will be thrown if the communicator is not set up correctly.

Communicator<B, T, D, A>\* communicatorPointer;

All four displayable entities are encapsulated in this Playable Object:

Displayable<CR\_ARRAY\_SIZE, T> cReferenceDisplayable;

Displayable<BR\_ARRAY\_SIZE, T> bReferenceDisplayable;

Displayable<PLANE\_ARRAY\_SIZE, T> planeDisplayable;

Displayable<DIGIT\_ARRAY\_SIZE, T> digitDisplayable;

The Displayable Object only stores the pixel coordinates, but it would be difficult to infer its relative center position and its current state. The following properties store the current state, such as the center position of the plane planeCenter and reference grids in both scopes bReferenceCenter and cReferenceYCenter, how close are two lines in the B-Scope reference bReferenceDistance, and the targeted position for random auto-movement for reference grids in both scopes bScopeTargetedShift and cScopeTargetedShift. They are internal helper properties used in game operation, and should never be explicitly read and modified by other objects.

T planeCenter[2];

T bReferenceCenter[2];

T cReferenceYCenter;

T cScopeTargetedShift;

T bReferenceDistance;

T bScopeTargetedShift[2];

The game requires concepts of finite state machine, and some enumeration-based properties are defined and used in the *processInputAndGenerateOutput()*function:

AttackState bScopeBeingAttacked;

DigitState digitState;

ScopeMode currentScopeMode;

Some internal clocks are redefined and used in *processInputAndGenerateOutputt()*function:

T automoveTimer;

T digitDisplayTimer;

T bScopeAttackTimer;

Each displayable entity has a unique pixel initialization function that provides a displayable entity specific wrapper function that eliminates the complexity of determining the correct argument for *initializePixelInformation()*from Displayable Object. Instead of manually determining all the line segments needed and their horizontal and vertical coordinates interval, these four functions are able to automatically initialize the correct pixel information.

For example, initializeCReferenceDisplayable() initializes cReferenceDisplayable by passing a single vertical center coordinate, initializeBReferenceDisplayable()initializes bReferenceDisplayable by passing a vertical center coordinate, a horizontal center coordinate, and a desired distance between two lines, initializePlaneDisplayable() initializes planeDisplayable by passing a vertical center coordinate and a horizontal center coordinate, and lastly *initializeDigitDisplayble*()initializes digitDisplayable by passing a vertical center coordinate, a horizontal center coordinate, and a desired character to be displayed.

bool *initializeCReferenceDisplayable*(T);

bool *initializeBReferenceDisplayable*(T, T, T);

bool *initializePlaneDisplayable*(T, T);

bool *initializeDigitDisplayble*(T, T, char);

At the same time, the difficulty level and the score that the player currently hold is stored too:

unsigned char difficultyLevel;

unsigned char score;

Internal functions are defined to eliminate repetitive code usages. These functions, however, shall not be called outside of Playable Object. A simple internal setter function *setCommunicator()* sets the Communicator pointer to communicatorPointer and performs null checks. *flashAllDisplayable*()flashes all four displayable entities to the screen exactly once. *processInputAndGenerateOutput*()is the most rigorous function in this entire project, which processes the game play by reading the input buffers from the Communicator Object, process the information received, and update the output buffers from the Communicator Object:

bool *setCommunicator*(Communicator<B, T, D, A> \*);

bool *flashAllDisplayable*();

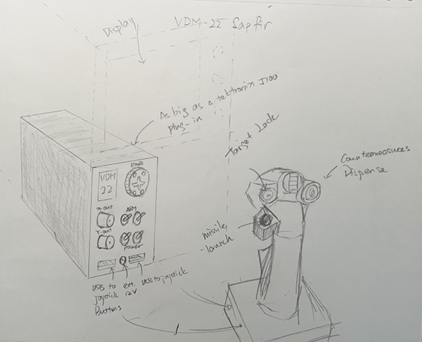
bool *processInputAndGenerateOutput*();

Finally, *executeCycle*()is the main event loop that should be called outside of Playable Object to make the game running.

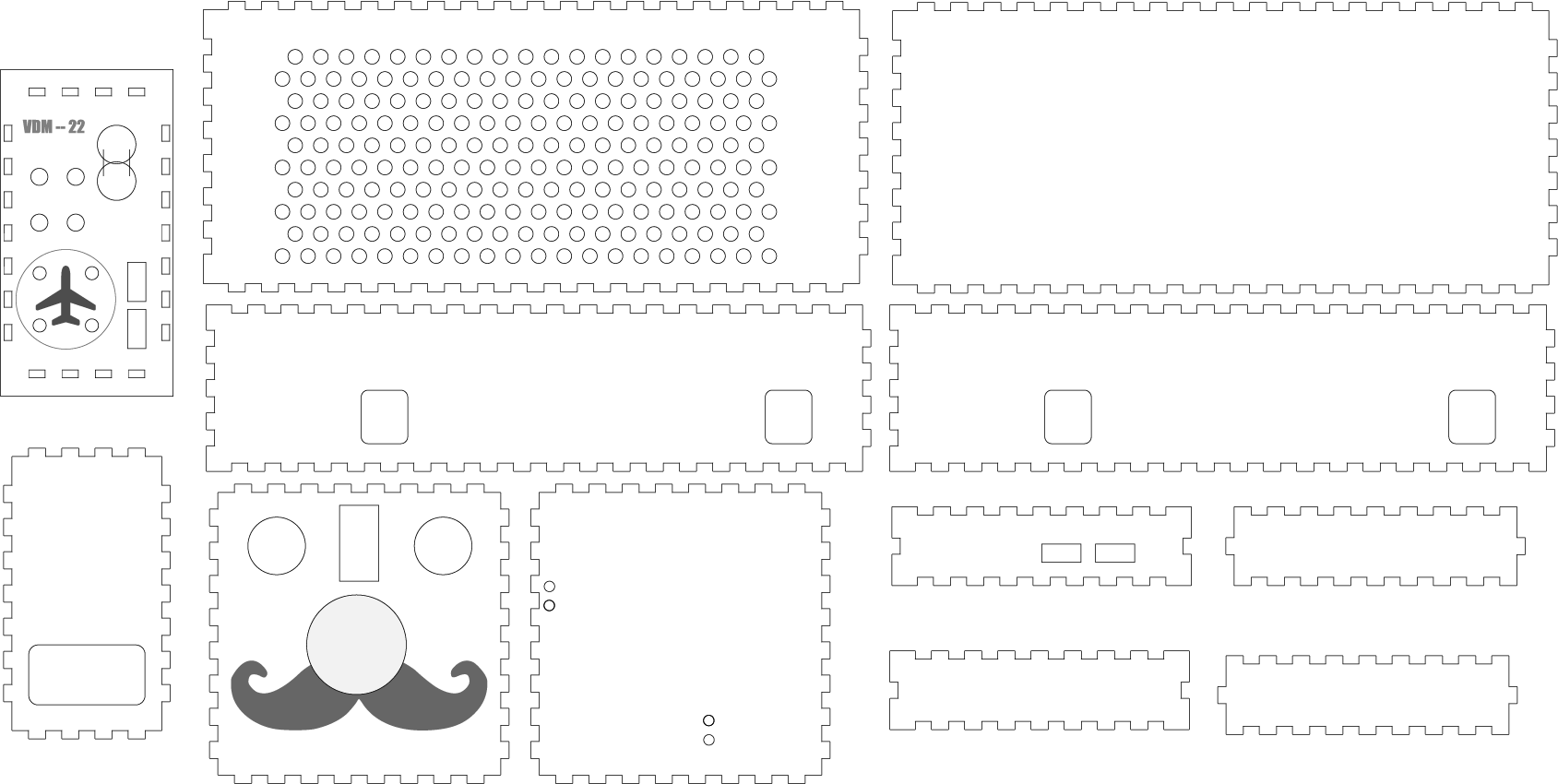
bool *executeCycle*();

## Enclosure Design Implmentation

As we are trying to make a box that has all the electronics part in it, we decided to use laser cutting techniques to design a simple box-like kits. Wood and plastic are used. We used pencil and paper to primarily design what the done looks like:



A model is subsequently constructed with details:



## Assembly and System Integration

From our previous experience with PCB designing and assembly, we decided it’s better to use through-hole components unless the design absolutely demands surface mount (massive integrations of multiple ICs or space constraint). Our PCB is completely through hole with the exception of the reset button, this made the board extremely easy to solder for our seasoned hardware engineer – Plug component in, pull out solder and iron, solder the component, great success!

No modification is required for debugging purposes. The serial headers and the Processor Direct Slot is included on board for easy of troubleshooting. Plus, the output being an analog oscilloscope also provided us with, well, an excellent oscilloscope for troubleshooting.

## Design Testing

### Testing Plan

The testing plan is fairly simple, power on the board and see if we get the expected output corresponding to our input. If not, use the excellent analog oscilloscope to probe the I/O pins for discrepancies.

### Testing Procedures

Two hardware issues were uncovered during the testing phase – after all, if the board works perfectly the first time it was plugged in, it wouldn’t be a Damienware™ (Wink Wink).

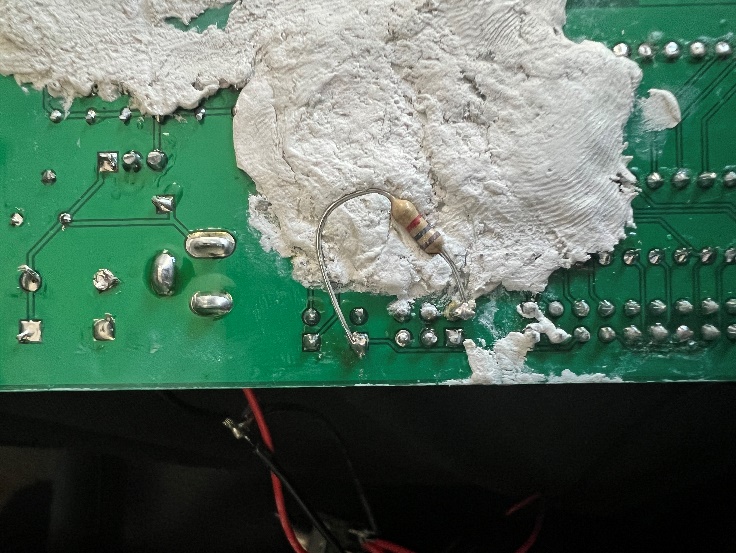
**The first problem** arrived when we discovered the joystick is not working as intended, or not working at all. While the X/Y axis is feeding the correct voltage level to our two ADC pins on the ATMega, the display always constantly moves to the left as if the X/Y axis are maxed out.

Our excellent software engineer then came up with an idea to test the input buffer of the ADC by using different pitches from the Piezo beeper. After a few rounds, we were able to determine that the ATMega were indeed reading in constant maximum value except when the joystick outputs 0 volts (minimum position). This discovery directly lead to our hardware engineer discovering that the AVREF (Analog Voltage Reference) was tied to GND. Our hardware engineer then remembered he may or may not consumed too much beer the night he was designing the microprocessor connection…

The problem were then promptly solved by first scratching out the AVREF pin’s connection to GND, and jumped over to VCC next doors. The fix is indeed successful and our joystick functions came back. Unfortunately, no pictures were taken at the night when this was fixed, and now it is coved up by insulating/adhering epoxy so we are unable to obtain a picture.

**The second problem** arrived when we discovered that our hardware engineer implemented the launch authority cutoff toggle switch without a corresponding pull-up resistor, causing the input pin to go floating when the launch authority switch is off. This was designed the same night our hardware engineer had a tad bit much beer…

The fix is again straightforward. The pull-up resistor next to the I/O region was cut and replaced with a jumper. The resistor is then connected between Vcc and the controller input pin where the launch authority pin headers reside. This time we took a picture:



The white substance next to the resistor is the insulating/adhering two-part epoxy used to adhere the board permanently to the enclosure.

Now, to see our design functioning in its full glory, please refer to the short movie *81192*, available today on [(132) Vector Drawing Machine 22 - YouTube](https://www.youtube.com/watch?v=B7cTL-qhFzI&t=217s)

## Budget and Cost Analysis

Our hardware engineer took special attention to not exceed two layers in this design, so the combined PCB cost is relatively low at 4975.4 USD, with only $0.497 per complete set of PCB (Processor board, RWR board and joystick board).

The components, however, is a different story. As the initial goal of our design is an extremely high-quality, low quantity product, our hardware engineer picked the best static and dynamic components in the industry. After calculations, the component cost for each PCB comes down to around $75, with the majority being the two high-quality copper-core BNC connectors for analog output. With Digikey’s bulk purchase discount, we managed to drop this price to around $50 per board. This means for 10,000 products, the component cost is at $500,000

The enclosure is entirely made of wood and acrylic, the cost of such material around $0.5 per product. This means we have a total cost of $509,975.4, at $51 per board. Given how expensive new stock retro computing equipment has become, our hardware engineer considers this a very reasonable price to pay for an entirely reprogrammable vector graphics machine.

## Team

Our team is made up of three members: Yinhao Qian, Shaoyu Pei and Damien Hu. Yinhao is probably the best software engineer in the ECE department, without his tremendous effort, our project would have never been possible**\***. Shaoyu is our enclosure engineer; his work ensured the VDM-22 eventually fit perfectly in the Tektronix 5110 oscilloscope. It almost looks like it is a purposefully made plug-in for the scope. Damien is our hardware engineer. The dozens of IBM/Compaq schematic read before this design certainly helped a lot. Computer assisted design didn’t exist in the 70s and early 80s. By absorbing the absolute genius of computer engineers during this era, the design of VDM-22 came to fruition.

The team started a Jira project together, but communications were done mainly through a WeChat group and weekly in-person catch up. As we are not actual full-time engineers, a waterfall model is implemented with set check off points for all three of us. This worked out really well considering all of us had something to do at all times.

**\**Clarification: It is not Yinhao who wrote this part.***

**Timeline**

Unfortunately, we do not have a timeline detailed to specific dates for this project, as we did the project on the go. However, a general milestone is：

Software: Visualizer UI Finished -> Library Finished -> Porting Finished

Hardware: PCB Design Finished -> Verification Finished -> Soldering Finished

Enclosure: Model Design Finished -> Laser Cutting Finished

Integration: Mounting and Software Uploading Finished -> Project Finished

## Summary, Conclusions and Future Work

We consider this project to be an absolute success. The team was able to integrate quite nicely. Having an expert in each field of the design (Hardware, Software, Enclosure) made the design process rather easy.

The second iteration will mainly focus on revamping the enclosure to include a latching mechanism to keep the enclosure in the scope. An adapter board is also in the works to power the VDM-22 directly from the system bus of the Tektronix 5110 oscilloscope.

1. The schematic can be found at the end of the document [↑](#footnote-ref-1)