Goal:

- Introduce design concepts in I/O operations and display interfaces.

Features to design in our CSOPESY emulator

- A means to support keyboard input, both real-time and event-driven.
  - Real-time = user can always type characters and this displays on-screen immediately. E.g. Word processor programs.
  - Event-driven = wait for an event, such as an enter command, and then perform an associated operation.
- How an application must behave under real-time/event-driven conditions.
  - Real-time = screen always refreshes even if there's no event/user input.
    - Sample applications: games, interactive applications like streaming websites.
    - Sample OSes: Modern OS like Windows are considered real-time.
  - Event-driven = screen refreshes after user input.
    - Sample applications:
    - Sample OSes: console/command line interfaces. However, there are means to support real-time switching. Example: Concept of tmux**,** pip download interface.

Examples of real-time applications

```cpp
#include <iostream>
#include <conio.h>
#include <Windows.h> // For Sleep()

void keyboardInterruptHandler() {
    if (_kbhit()) {
        char key = _getch();
        std::cout << "Key pressed: " << key << std::endl;
    }
}

int main() {
    while (true) {
        // Simulate other program logic
        Sleep(100); // Sleep for 100 milliseconds

        // Check for keyboard input through the interrupt handler
        keyboardInterruptHandler();
    }

    return 0;
}
```

The example above shows a real-time keyboard polling mechanism.

Another example in CSOPESY emulator, where it shows the main loop. A real-time application would typically have a while loop wherein the application exits the loop when an exit/terminating sequence is triggered.

```cpp
int main()
{
    InputManager::initialize();
    FileSystem::initialize();
    // FileSystem::getInstance()->test_createRandomFiles(1000);
    // FileSystem::getInstance()->saveFileSystem();
    FileSystem::getInstance()->loadFileSystem();
    ConsoleManager::initialize();
    MessageBuffer::initialize();
    ResourceEmulator::initialize();
    MemoryManager::initialize();

    bool running = true;
    while(running)
    {
        ConsoleManager::getInstance()->process();
        ConsoleManager::getInstance()->drawConsole();

        running = ConsoleManager::getInstance()->isRunning();
    }

    // MemoryManager::test_MemoryAllocation();

    MemoryManager::destroy();
    ResourceEmulator::destroy();
    MessageBuffer::destroy();
    ConsoleManager::destroy();
    InputManager::destroy();
    return 0;
}
```

Another example in a game application where real-time interaction is highly valued:

```cpp
// Game Loop
void RunGameLoop(IGameEvent& gameEvent) {
    const float targetFPS = 60.0f;
    const float frameTime = 1.0f / targetFPS;

    while (true) {
        // Handle user input
        if (_kbhit()) {
            char key = _getch();
            gameEvent.OnKeyPressed(key);
        }

        // Update game logic
        gameEvent.OnUpdate(frameTime);

        // Render game state
        gameEvent.OnRender();

        // Sleep to achieve a constant frame rate
        Sleep(static_cast<DWORD>(frameTime * 1000.0f));
    }
}

int main() {
    GameEventHandler gameHandler;

    // Start the game loop
    RunGameLoop(gameHandler);

    return 0;
}
```

Examples of event-driven applications
Event-driven applications are those in which the flow of the program is determined by events such as user actions, sensor outputs, or messages from other programs

- Graphical User Interface (GUI) Applications
    - Example: A simple text editor
    - Events: Mouse clicks, keypresses, menu selections
    - Usage: When the user interacts with the GUI by clicking buttons, typing text, or choosing options from menus, events are triggered, and the application responds accordingly.

The code below shows Qt framework in C++, designed for event-driven applications:
https://www.qt.io/product/framework

```cpp
//The code below shows Qt framework in C++, designed for event-driven applications.
#include <QApplication>
#include <QTextEdit>
#include <QMenuBar>
#include <QMenu>
#include <QAction>
#include <QFileDialog>

class TextEditor : public QTextEdit {
    Q_OBJECT

public:
    TextEditor(QWidget *parent = nullptr) : QTextEdit(parent) {
        createActions();
        createMenus();
    }

private:
    void newFile() {
        clear();
    }

    void openFile() {
        //open file
    }

    void saveFile() {
        //save file
    }

private:
    void createActions() {
        newAction = new QAction(tr("&New"), this);
        newAction->setShortcut(QKeySequence::New);
        connect(newAction, &QAction::triggered, this, &TextEditor::newFile);

        openAction = new QAction(tr("&Open"), this);
        openAction->setShortcut(QKeySequence::Open);
        connect(openAction, &QAction::triggered, this, &TextEditor::openFile);

        saveAction = new QAction(tr("&Save"), this);
        saveAction->setShortcut(QKeySequence::Save);
        connect(saveAction, &QAction::triggered, this, &TextEditor::saveFile);
    }

    void createMenus() {
        fileMenu = menuBar()->addMenu(tr("&File"));
        fileMenu->addAction(newAction);
        fileMenu->addAction(openAction);
        fileMenu->addAction(saveAction);
    }

    QMenu *fileMenu;
    QAction *newAction;
    QAction *openAction;
    QAction *saveAction;
};

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    TextEditor textEditor;
    textEditor.show();

    return app.exec();
}
```

Web Applications

- Example: An online shopping website
- Events: Button clicks, form submissions, AJAX requests
- Usage: Users interact with the website by clicking buttons to add items to their cart, submitting forms for payment, and the application responds to these events.

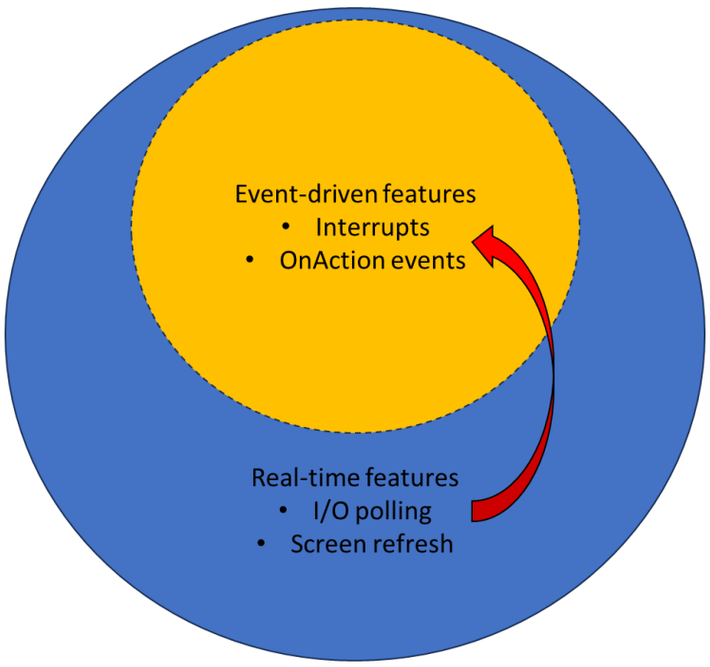
Mobile Applications:

- Example: A social media app
- Events: Taps, swipes, device orientation changes
- Usage: Users interact with the app by tapping on posts, swiping to navigate, and the application responds by displaying relevant content.

Networking Applications:

- Example: Chat application
- Events: Receiving a message, connection status changes
- Usage: When a user sends a message, or when the application receives a message from another user, events are triggered, and the application updates the chat interface.

Observation: All event-driven applications, are derived from real-time applications. All event-driven applications still require functionality that needs to run in real-time.



Family of implementation features

Examples of this observation
A background in hardware-level I/O systems.

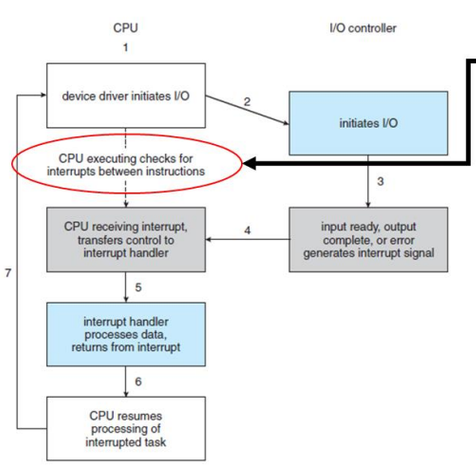
All hardware have an **interrupt mechanism**.

- In an interrupt-driven system, the CPU is interrupted when a key is pressed.
- The operating system or a low-level handler responds to the interrupt and takes appropriate action → a system API call could be exposed where events can be handled by the developer. E.g. onKeyDown(), onKeyUp().
- This method is more event-driven and can be more efficient as the CPU is not constantly checking for input.

The question then is → how do we check for any I/O events?

12.2.3 Interrupts

The basic interrupt mechanism works as follows. The CPU hardware has a wire called the interrupt-request line that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt-request line, the CPU performs a state save and jumps to the interrupt-handler routine at a fixed address in memory. The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a `return from interrupt` instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller *raises* an interrupt by asserting a signal on the interrupt request line, the CPU *catches* the interrupt and *dispatches* it to the interrupt



Real-time → constantly checks for interrupts per cycle (after each instruction)

```
68  //Real-time --> constantly checks for interrupts per cycle (after each instruction)
69  while(running) {
70      execute();
71      bool flag = hasIORequests();
72
73      if(flag) {
74          <do something with IO requests>;
75      }
76  }
```

The CPU hardware has an interrupt-request line → the interrupt-request line is coded in such a way that it efficiently checks for any I/O requests.

Consider Windows API. The WndProc is the window instance that contains system-level calls for I/O requests. But first, we need to have a **message loop:**

```
MSG msg;
while (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

- Windows applications often have a message loop that continually checks for and dispatches messages.
- The GetMessage() function is commonly used to retrieve messages from the application's message queue.

Callbacks are provided in the WndProc handle.

```
78  //wndproc callback sample
79  LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
80      switch (msg) {
81          case WM_KEYDOWN:
82              // Handle keydown event
83              break;
84          case WM_MOUSEMOVE:
85              // Handle mouse movement event
86              break;
87          // Other cases for different messages...
88          default:
89              return DefWindowProc(hwnd, msg, wParam, lParam);
90      }
91      return 0;
92  }
```

- Each window has an associated window procedure (WndProc) that handles messages for that window.
- Messages related to user input, such as keyboard and mouse events, are processed in the window procedure.

Input events could be registered in the RegisterRawInputDevices() function.

```
RAWINPUTDEVICE Rid[1];
Rid[0].usUsagePage = 0x01;
Rid[0].usUsage = 0x06; // Keyboard
Rid[0].dwFlags = 0;
Rid[0].hwndTarget = hwnd;
RegisterRawInputDevices(Rid, 1, sizeof(Rid[0]));
```

By using these mechanisms, developers can handle input events in a Windows application without directly dealing with low-level interrupts. The Windows operating system itself takes care of interrupt handling and provides a higher-level API for application developers.

Consider **Qt framework**, and other event-driven APIs available in C++/Java → who throws the OnKeyUp(), OnKeyDown() events?

Take-away: Even at its lowest level (hardware and OS), real-time features are implemented and event-driven features are thrown, from real-time events detected.

Here is an example of a simple C++ application that combines real-time and event-driven handling:
Real-time feature → keyboard polling,
Event-driven feature → IKeyboardEvent implementation

```cpp
//Here is an example of a simple C++ application that combines real-time and event-driven handling:
#include <conio.h> // For _kbhit() and _getch()

// Interface for Keyboard Events
class IKeyboardEvent {
public:
    virtual void OnKeyDown(char key) = 0;
    virtual void OnKeyUp(char key) = 0;
};
// Implementation of IKeyboardEvent
class KeyboardEventHandler : public IKeyboardEvent {
public:
    void OnKeyDown(char key) override {
        std::cout << "Key Down: " << key << std::endl;
    }

    void OnKeyUp(char key) override {
        std::cout << "Key Up: " << key << std::endl;
    }
};
// Keyboard Polling
void PollKeyboard(IKeyboardEvent& keyboardEvent) {
    while (true) {
        if (_kbhit()) { // Check if a key has been pressed
            char key = _getch(); // Get the pressed key

            // Check if it's a key down or key up event
            if (GetAsyncKeyState(key) & 0x8000) {
                keyboardEvent.OnKeyDown(key);
            } else {
                keyboardEvent.OnKeyUp(key);
            }
        }

        // Other program logic can continue here
    }
}

int main() {
    KeyboardEventHandler keyboardHandler;
    // Start keyboard polling
    PollKeyboard(keyboardHandler);
    return 0;
}
```

- The IKeyboardEvent class defines an interface with OnKeyDown and OnKeyUp virtual functions.
- The KeyboardEventHandler class implements this interface, printing messages to the console when a key is pressed or released.

- The PollKeyboard function continuously checks for key presses using _kbhit() and retrieves the key using _getch().
- Depending on the state of the key using GetAsyncKeyState, it calls the appropriate event handler.

DISPLAY INTERFACE

- How should we draw on the screen? The goal is to support **both** real-time and event-driven scenarios.
- Thus, for every cycle, we must have mechanisms to refresh the screen → extending the screen refresh, such that it will only redraw after a user event is easy to do → recall OS interrupt mechanism.

```
bool running = true;
while(running)
{
    ConsoleManager::getInstance()->process();
    ConsoleManager::getInstance()->drawConsole();

    running = ConsoleManager::getInstance()->isRunning();
}
```

- process() should contain handling of logic and other non-drawing operations.
- DrawConsole() refreshes the screen with the updated information.

Each console window must have the following implementations:

```
class AConsole
{
public:
    typedef std::string String;
    AConsole(String name);
    ~AConsole() = default;

    String getName();
    virtual void onEnabled() = 0;
    virtual void display() = 0;
    virtual void process() = 0;

    String name;
    friend class ConsoleManager;
};
```

- onEnabled() is called every time the console first draws to the screen.
- process() is called every frame
- display() is called every frame

Temporary and easy solution to creating an event-driven console: Put the drawing in the process() method, and use std::in to wait for user input in order to pause the screen refresh.

TODO: show the implementation of the following from the CSOPESY-Emulator code:

- AConsole class
- ConsoleManager

Activities

- Create a means to support multi-window **real-time** drawing of screens.
    - Using the screen -s <name> format, create custom screens: screen1, screen2, screen3, where each screen is created by an individual in your student group.
    - Study the Windows console API → how to move your cursor, set console window size, and set font color.
    - Continue the “marquee” command wherein it moves the marquee screen once recognized. The marquee screen simply displays “Hello, marquee”, and then the user can type “exit” to go back to the main menu.
    - Create your own ASCII-animated screen wherein the animation speed is controlled by a refresh rate.

      ```
      static constexpr int REFRESH_DELAY = 10; //screen refresh in marquee console
      static constexpr int POLLING_DELAY = 5; //keyboard polling rate. Lower is better.
      ```

    -
    - Link everything together in your main menu.
    - Be prepared to be called in class to discuss your implementation.