Extron ControlScript Project

Project Overview

This is an Extron ControlScript project for controlling AV equipment using Python-based programming instead of Extron's traditional GUI configuration software.

Project Name: extron **Author:** Euan Murray (e.l.murray@ed.ac.uk) **Version:** 0.0.1 **System ID:** 4787

Important: Python 3 Environment

Extron ControlScript uses Python 3 (specifically Python 3.11.8 on Pro xi controllers). Key considerations:

- F-strings (`f'text {var}'`) are supported but may cause issues with some parsing - use `.format()` for safety
- Type hints are supported
- All standard Python 3 features available
- Memory management is important on embedded devices

Deployment Scope and Requirements

This project is designed for **large-scale multi-room deployment** with the following requirements:

Scale

- **150 rooms** to configure and deploy
- Each room has unique hardware configuration (controller and touchpanel types)
- Single codebase must support all room variations
- Deployment via Extron ControlScript Deployment Utility

Standardization

- **Shared GUI file(s)** - All rooms use the same touchpanel layout(s)
- **Fixed button IDs** - UI elements have consistent IDs across all rooms
- **Modular drivers** - Each room loads only the device modules it needs
- **Room-specific configuration** - Unique JSON descriptor per room (like `newhaventlt.json`)

Architecture Goals

1. **Single source code repository** - All rooms share the same Python codebase
2. **Configuration-driven deployment** - Room differences handled via JSON descriptors
3. **Scalable and maintainable** - Easy to update all rooms or individual rooms
4. **Modular device drivers** - Dynamic loading based on room configuration
5. **Standardized UI** - Consistent user experience across all rooms

Hardware Configuration (Example Room: Newhaven LT)

This section describes one example room configuration. Each of the 150 rooms will have its own JSON descriptor with different hardware.

Primary Processor

- **Model:** IPCP Pro 255Q xi
- **Part Number:** 60-1914-01
- **Alias:** Processor1
- **IP Address:** 172.19.31.109
- **ControlScript Version:** Pro xi (1.8.21xi recommended)

Touch Panel

- **Model:** TLP Pro 1025T
- **Part Number:** 60-1565-02
- **Alias:** TLP1
- **IP Address:** XXX.XX.XX.XXX
- **Layout File:** TLP Central Group.gdl
- **Extron Control Web ID:** tlp15scsnewhaven

Room Variation Examples

Across the 150 rooms, we expect variations in:

- Processor models (IPCP Pro 255Q xi, IPCP 555, IPCP Pro 555, etc.)
- Touchpanel models (TLP Pro 1025T, TLP Pro 725T, TLP Pro 520M, etc.)
- Connected devices (displays, projectors, DSPs, matrix switchers, etc.)
- Serial/IP controlled equipment requiring different driver modules

Project Structure

```
extron/
├── src/                        # Shared source code (deployed to all rooms)
│   ├── main.py                 # Program entry point - imports and initializes system
│   ├── config.py               # Configuration loader (reads room-specific settings)
```

```
│   ├── devices.py              # Device definitions (dynamically loaded)
│   ├── variables.py            # Global variables and data
│   ├── system.py               # System logic, automation, and Initialize()
│   ├── ui/
│   │   └── tlp.py             # Touch panel UI objects and event handlers
│   │       └── common.py       # Common UI functions shared across all rooms
│   ├── control/
│   │   └── av.py              # AV control logic (base/common functionality)
│   └── drivers/                # Device driver modules (third-party equipment)
│       ├── display/
│       ├── projector/
│       ├── dsp/
│       └── switcher/
├── modules/
│   └── helper/
│       └── ModuleSupport.py    # Helper utilities (including eventEx decorator)
├── helpers/                    # Helper modules and interfaces
│   └── gve_interface_v2x2x0/
├── layout/                     # Touch panel layout files (.gdl) - SHARED
├── sound/                      # Audio files for UI feedback - SHARED
├── rfile/                      # Resource files - SHARED
├── ir/                         # IR command files
├── configs/                    # Room-specific configuration files
│   ├── newhaventlt.json       # Example room configuration
│   ├── room_001.json
│   ├── room_002.json
│   └── ... (150 total)
├── docs/                       # Documentation (copied from VSCode extension)
│   ├── extension-help/         # VS Code extension help documentation
│   ├── api/                    # ControlScript API documentation (all versions)
│   └── architecture/           # Architecture decision records
│       └── MULTI_ROOM_OPTIONS.md # Options analysis for multi-room deployment
└── README.md                   # This file
```

Key Structure Notes

Shared Code (`src/`):

- All Python code is shared across all rooms
- Uses configuration-driven approach to handle room differences
- Dynamically imports device drivers based on room configuration

Room-Specific Files (`configs/`):

- Each room has a unique JSON descriptor
- Deployment utility uploads appropriate JSON for each room
- JSON specifies: processor, touchpanel, devices, IP addresses, driver modules

Shared Resources (`layout/` , `sound/` , `rfile/`):

- GUI layouts standardized across all rooms (fixed button IDs)

- Audio and resource files shared
- Reduces deployment size and ensures consistency

Documentation

All Extron ControlScript documentation has been copied locally to the `docs/` folder for easy reference and offline access.

Extension Help Documentation

Location: `docs/extension-help/`

The main help system for the Extron ControlScript VS Code extension. This includes:

- Getting started guides
- Project structure best practices
- Snippets reference
- Template manager documentation
- Data file integration
- Version switcher information

Main Entry Point: `docs/extension-help/ControlScript_Extension_for_VS_Code_Help.htm`

API Documentation

Location: `docs/api/`

Complete API reference documentation for all ControlScript versions:

Pro xi Versions (High-end processors like IPCP Pro series)

- `docs/api/1.8.21xi/` - **Latest Pro xi** (Recommended for this project)
- `docs/api/1.7.10xi/`
- `docs/api/1.6.10xi/`
- `docs/api/1.5.4xi/`

Standard Versions (Standard processors like IPCP 255, 505)

- `docs/api/3.13.39/` - Latest Standard
- `docs/api/3.12.5/`
- `docs/api/3.11.4/`
- `docs/api/3.10.4/`

Note: Since this project uses an IPCP Pro 255Q xi processor, the **1.8.21xi API documentation** is the most relevant.

Key Concepts

extronlib Package

The `extronlib` package is Extron's Python library that provides all control functionality:

- `extronlib.device` - Control devices (ProcessorDevice, UIDevice, SPDevice)
- `extronlib.interface` - Hardware interfaces (Serial, Relay, DigitalIO, Ethernet, etc.)
- `extronlib.ui` - User interface objects (Button, Label, Slider, Knob, etc.)
- `extronlib.system` - System utilities (Wait, ProgramLog, MESet, etc.)

Project Flow

1. **main.py** - Entry point that imports all modules and calls `system.Initialize()`
2. **devices.py** - Defines all hardware devices and interfaces
3. **variables.py** - Sets up global data and state
4. **ui/tlp.py** - Creates UI objects and button event handlers
5. **control/av.py** - Implements control logic for AV equipment
6. **system.py** - Connects devices, starts services, and initializes the system

Project Descriptor

File: `newhaventlt.json`

This JSON file defines the system configuration including:

- System metadata (name, author, version)
- Device list with part numbers and network configuration
- File paths for code, layouts, sounds, and resources
- UI settings for touch panels

Getting Started

Prerequisites

- Visual Studio Code
- Extron ControlScript Extension (v1.9.0-16 or later)
- Python 3.x (for local development/testing)
- Network access to Extron devices

VS Code Extension Features

- **Command Palette:** Press F1 and type "Extron:" to see all commands

- **Version Switcher:** Click "ControlScript" in status bar to switch API versions
- **Snippets:** Type "Extron" to see available code snippets
- **Control System Tree:** View project devices in VS Code Explorer
- **IntelliSense:** Full autocomplete for extronlib objects and methods

Common Snippets

- `Extron Button Instantiation` - Define a button
- `Extron Event Button Pressed` - Button press handler
- `Extron Interface SerialInterface` - Create serial interface
- `Extron Device Processor` - Add processor device
- Full list available in extension help docs

Development Workflow

For Multi-Room Deployment

1. **Design Shared GUI** - Create standardized touchpanel layout with fixed button IDs
2. **Develop Core Code** - Write shared Python code in `src/` that works for all rooms
3. **Create Device Drivers** - Build modular drivers in `src/drivers/` for different equipment
4. **Configure Rooms** - Create JSON descriptors for each of 150 rooms in `configs/`
5. **Test on Reference Room** - Deploy and test on one room first
6. **Batch Deploy** - Use Deployment Utility to upload to all rooms with appropriate JSON

For Single Room Development

1. **Edit Project Descriptor** - Add/modify devices in room's JSON file (e.g., `configs/newhaventlt.json`)
2. **Define Devices** - Update `devices.py` to dynamically load based on configuration
3. **Create UI Objects** - Define buttons, labels, etc. in `ui/tlp.py` using fixed IDs
4. **Add Event Handlers** - Implement button press/release events in `ui/tlp.py`
5. **Implement Control Logic** - Add AV control code in `control/av.py`
6. **Add Driver Support** - Create/import device drivers as needed in `src/drivers/`
7. **Initialize System** - Connect devices and start services in `system.Initialize()`
8. **Deploy to Processor** - Use VS Code extension or Deployment Utility to upload

Multi-Room Deployment Strategies

Challenges

1. **Device Diversity** - 150 rooms with different hardware configurations
2. **Code Reusability** - Single codebase must work across all room types
3. **Maintainability** - Updates should deploy easily to all or specific rooms

4. **Configuration Management** - 150 JSON files need to be managed and version controlled
5. **Driver Loading** - Different rooms need different third-party device drivers

Architectural Options (Under Investigation)

See `docs/architecture/MULTI_ROOM_OPTIONS.md` for detailed analysis.

Option 1: Configuration-Driven Dynamic Loading

- Single codebase with room-specific JSON configuration
- Dynamically import device drivers based on JSON metadata
- Conditional logic based on device presence in configuration
- Pros: Most flexible, easiest to maintain
- Cons: Requires careful error handling, more complex initialization

Option 2: Template-Based Project Generation

- Use Extron Template Manager to generate room-specific projects
- Each room is a separate project with customized code
- Pros: Simple, uses built-in Extron tooling
- Cons: 150 separate codebases, difficult to update globally

Option 3: Hybrid Approach

- Core functionality in shared modules
- Room-specific configuration + data files
- Minimal room-specific code overrides
- Pros: Balance between flexibility and maintainability
- Cons: More complex project structure

Option 4: Feature Flag System

- All possible device drivers included in codebase
- Configuration enables/disables features per room
- Pros: Simplest deployment, no dynamic loading
- Cons: Larger code footprint, potential memory concerns

Recommended Approach (TBD)

Further investigation needed to determine best approach based on:

- Extron best practices documentation
- Memory constraints of smallest processor
- Complexity tolerance
- Deployment tooling capabilities

Global Viewer Enterprise (GVE) Integration

Overview

Global Viewer Enterprise (GVE) is Extron's cloud-based monitoring and management platform. Integration with GVE is **CRITICAL** for this project because it provides centralized monitoring of all 150 rooms from a single dashboard.

Why GVE is Essential

Managing 150 rooms manually is impractical. GVE provides:

1. **Centralized Monitoring**
   - View status of all 150 rooms from one interface
   - Real-time device health monitoring
   - Connection status for all equipment
2. **Proactive Maintenance**
   - Monitor projector lamp hours across all rooms
   - Alert staff before equipment fails
   - Track filter hours and usage statistics
   - Reduce unexpected equipment failures
3. **Remote Troubleshooting**
   - Diagnose issues without visiting the room
   - Test equipment remotely (power on/off, source switching)
   - Reduce mean time to resolution
4. **Usage Analytics**
   - Track equipment utilization across campus
   - Inform purchasing decisions
   - Plan maintenance schedules
   - Generate usage reports
5. **Staff Efficiency**
   - Reduce on-site visits
   - Prioritize maintenance tasks
   - Respond faster to issues
   - Single pane of glass for entire AV infrastructure

GVE Helper Module

Location: `helpers/gve_interface_v2x2x0/` **Version:** 2.2.0 **Copyright:** 2015-2022, Extron

The GVE integration is provided via an Extron helper module that handles:

- TCP/UDP communication with GVE server
- Automatic heartbeat (every 20 seconds) to maintain connection

- Protocol implementation (GS-to-GVE interface)
- Event-driven architecture for status updates and remote commands
- MAC address-based device identification

Integration Requirements

All 150 rooms must integrate GVE monitoring:

1. **Initialize GVE Client**

```
from helpers.gve_interface_v2x2x0.src.gve_interface import gveClient

# Initialize connection to GVE server
GVEServer = gveClient('gve.institution.edu', MainProcessor, IPPort=5555)
```

2. **Assign Device IDs**
   - Each monitored device needs a unique GVE device ID
   - IDs are assigned in the GVE server configuration
   - Room configuration JSON maps devices to GVE IDs
3. **Report Device Status**

```
# Send device status to GVE
GVEServer.SendStatus('Proj001', 'Power', 'On')
GVEServer.SendStatus('Proj001', 'Connection', 'Connected')
GVEServer.SendStatus('Proj001', 'Lamp 1 Hours', '1250')
GVEServer.SendStatus('Proj001', 'Device Status', 'Normal')
```

4. **Handle Remote Commands**

```
# Receive and execute commands from GVE dashboard
@event(GVEServer, 'ReceiveGVECommand')
def GVEExecuteCommand(server, data):
    device, command, parameter = data
    # Execute remote command (power on/off, source select, etc.)
```

Monitorable Device States

GVE can monitor the following device states:

- **Power:** Unknown, Off, On, Cooling Down, Warming Up
- **Connection:** Unknown, Disconnected, Connected, Online, Offline
- **Source:** Current input selection (HDMI, DVI, VGA, etc.)
- **Lamp Hours:** Lamp 1-4 hours (for projectors)
- **Filter Hours:** Air filter runtime
- **Device Status:** Normal/Warning/Error with custom status messages

GVE Events

The GVE client provides these event handlers:

- `DiagnosticSendText` - Triggered when data sent to GVE server
- `DiagnosticReceiveText` - Triggered when data received from GVE
- `GVEServerConnected` - Connection to GVE established
- `GVEServerDisconnected` - Connection to GVE lost
- `ReceiveGVECommand` - Remote command received from GVE dashboard
- `ReceiveGVERoomEvent` - Room-level event triggered (SystemOn, SystemOff)

Configuration Approach

For scalable GVE integration across 150 rooms:

1. **Shared GVE Server** - All rooms connect to single GVE instance
2. **Unique Device IDs** - Each device in each room has unique ID
3. **Room Configuration** - GVE settings in room-specific JSON files
4. **Automatic Status Reporting** - Device drivers report status changes to GVE
5. **Remote Control Support** - Rooms respond to GVE commands

Example Configuration

```
{
  "gve": {
    "enabled": true,
    "server_address": "gve.institution.edu",
    "server_port": 5555,
    "device_mappings": [
      {
        "gve_device_id": "Proj001",
        "device_type": "projector",
        "friendly_name": "Main Projector",
        "monitor_states": ["Power", "Connection", "Lamp 1 Hours", "Filter Hours", "Device Status"]
      }
    ]
  }
}
```

Implementation Notes

- GVE integration must be part of core system initialization
- All device drivers should report state changes to GVE
- Device status must update when:
  - Device powers on/off
  - Connection established/lost

    - Source changes
    - Error conditions occur
- Heartbeat maintains connection automatically (no manual intervention)
- GVE room IDs assigned by server (cannot be manually set)

Documentation

See `docs/architecture/MULTI_ROOM_OPTIONS.md` section 6 for detailed GVE integration architecture and implementation patterns.

Resources

- **Local Documentation:** See `docs/` folder
- **Extron Website:** https://www.extron.com/
- **Support:** Contact Extron technical support or your Extron account representative
- **Training:** Extron offers online and in-person ControlScript training courses

License

Next Steps

Immediate Priorities

- [ ] Research Extron best practices for multi-room deployment
- [ ] Investigate data file and template management features
- [ ] Determine optimal architectural approach for 150-room scale
- [ ] Create proof-of-concept for configuration-driven device loading

Development Tasks

- [ ] Design standardized GUI layout with fixed button IDs
- [ ] Set up configs folder with example JSON descriptors
- [ ] Implement configuration loader module (config.py)
- [ ] Create driver framework for modular device control
- [ ] Build reference implementation for one room type

Deployment Planning

- [ ] Document Deployment Utility workflow
- [ ] Create room inventory spreadsheet (all 150 rooms)
- [ ] Plan phased rollout strategy
- [ ] Establish testing and validation procedures