🚀 실리카(Xilica) Lua 스크립트 실무/고급자 가이드

이 문서는 기초를 넘어 **복잡한 제어 로직, 외부 장비 통신(TCP/UDP), 프로토콜 파싱** 등 실무에서 마주치는 난제들을 해결하기 위한 고급 기법을 다룹니다.

🎖️ 1. 중급 (Intermediate): 효율적인 로직 설계

초보자가 `if-else`를 수없이 복사붙여넣기 할 때, 중급자는 **테이블(Table)**을 사용하여 코드를 단 몇 줄로 줄입니다.

🔷 테이블 매핑 (Table-Driven Logic)

버튼 10개가 각각 다른 명령어를 보낸다면? ❌ **하수 (If 지옥)**

```
if in_t[1] == 1 then cmd = "VOL_10" end
if in_t[2] == 1 then cmd = "VOL_20" end
...
```

✅ **고수 (Table 사용)** 명령어를 테이블에 미리 정의해두고, 반복문으로 처리합니다.

```
local CMD_MAP = {
    [1] = "VOL_10",
    [2] = "VOL_20",
    [3] = "MUTE_ON",
    [4] = "MUTE_OFF"
}

-- 루프 한 번으로 모든 버튼 처리
for pin, command in pairs(CMD_MAP) do
    if in_t[pin] == 1 then
        out_t[1] = command -- 바로 전송
        break -- 동시 입력을 막으려면 break
    end
end
```

🔷 상태 관리 (State Management)

버튼을 뗄 때도 신호가 발생합니다. **"눌렀을 때만(Rising Edge)"** 동작하게 하려면 이전 상태를 기억해야 합니다.

```
-- 전역 변수 초기화 (메모리)
if G_LastState == nil then G_LastState = {} end

local current_val = in_t[1]
local last_val = G_LastState[1] or 0

-- 이전에는 0이었는데 지금 1이 되었다면? (Rising Edge)
if last_val == 0 and current_val == 1 then
    out_t[1] = "TRIGGER!"
end
```

```
-- 상태 업데이트 (다음 틱을 위해)
G_LastState[1] = current_val
```

🏆 2. 고급/실무자 (Expert): 외부 장비 통신

실무의 꽃은 **타사 장비(Projector, Matrix, Sensor)**와의 통신입니다.

🔷 바이너리/Hex 프로토콜 처리

많은 산업용 장비(Projector, Novastar 등)는 텍스트가 아닌 **Hex 코드**를 요구합니다. Xilica Lua는 문자열이 바이너리를 담을 수 있습니다.

1. Hex 문자열 만들기

```
-- 예: 0x02 0x00 0x00 0x03 (STX, Data, ETX)
local hex_cmd = "\x02\x00\x00\x03"

-- 동적으로 계산해서 만들기 (string.char)
local vol = 100 -- 0x64
local dyn_cmd = "\x02" .. string.char(vol) .. "\x03"
```

2. 체크섬(Checksum) 계산 장비들은 데이터가 깨졌는지 확인하기 위해 체크섬을 요구합니다.

```
-- 체크섬 계산 예제 (단순 합계)
local payload = "\x01\x02\x64"
local sum = 0

-- 문자열을 한 글자씩 쪼개서 숫자로 변환 후 더하기
for i = 1, #payload do
    sum = sum + string.byte(payload, i)
end

-- 하위 8비트만 사용 (Modulo 256)
local checksum = sum % 256
local final_packet = payload .. string.char(checksum)
```

🔷 중앙 집중식 상태 추적 (Centralized State Tracking)

대규모 미러링 시스템이나 다채널 볼륨 제어에서 가장 중요한 패턴입니다. Lua를 단순한 스위치가 아닌 **"상태 관리자"**로 활용합니다.

패턴 이름: `InTable/OutTable State Tracking`

- **목적**: 수십 개의 fader/버튼 상태를 감시하고, 변화가 있을 때만 네트워크 명령어를 한 번씩 내보냄.
- **장점**: 네트워크 부하 최소화, TCP 소켓 안정성 확보.

```
-- Global 변수를 사용해 이전 상태를 기억
if Previous_State == nil then
    Previous_State = {} -- [핀번호] = 마지막값
end
```

```lua
for i = 1, 10 do
    local val = tonumber(in_t[i] or 0)

    -- 이전 값과 비교 (약간의 오차 허용)
    if math.abs(val - (Previous_State[i] or -999)) > 0.001 then
        -- 변화 발생! 명령어 생성
        out_t[1] = string.format("SET CH%d_VOL %.1f\r", i, val)
        Previous_State[i] = val
        break -- 한 번에 하나씩만 보내서 네트워크 정체 방지
    end
end
```

🔷 하트비트 (Keepalive) 패턴

TCP 연결이 끊기지 않게 주기적으로 신호를 보내는 **"자가 발전"** 로직입니다.

```lua
if G_Timer == nil then G_Timer = 0 end

-- 매 스캔 사이클(약 0.05~0.1초)마다 증가
G_Timer = G_Timer + 1

-- 약 10초마다 (사이클 타임에 따라 값 조절 필요)
if G_Timer > 200 then
    out_t[1] = "PING\r\n" -- 네트워크 포트로 PING 전송
    G_Timer = 0           -- 타이머 초기화
end
```

🛠️ 3. 디버깅 및 트러블슈팅 가이드

스크립트가 안 돌 때 체크해야 할 사항들입니다.

1. 닐(Nil) 포비아

Lua 오류의 90%는 `nil` 때문입니다.

- **증상**: 스크립트 멈춤, 연산 불가.
- **해결**: 모든 입력값에 "방탄 조끼"를 입히세요.

```lua
local val = in_t[1]
if val == nil then val = 0 end -- 방탄 코드
```

2. 가상 테스팅 (Mocking)

현장에 장비가 아직 없다구요? 가상의 응답을 만들어서 테스트하세요.

```lua
-- 장비에서 응답이 왔다고 치자 (시뮬레이션)
-- Network Client의 Rx 핀(예: 10번)에 연결된 것처럼
local rx_data = in_t[10]
```

```
if rx_data == "OK" then
    out_t[2] = 1 -- LED 켬
end
```

3. 사이클 타임 고려

Xilica Lua는 매우 빠르게 반복 실행됩니다.

- 무거운 `for` 문(10000번 반복 등)을 돌리면 전체 오디오 프로세서가 느려질 수 있습니다.
- 복잡한 로직은 최대한 간결하게, 필요할 때만 실행되도록 `if` 문으로 감싸세요.

이 가이드 라인을 따라 설계하시면, 단순한 버튼 연결을 넘어 **"지능형 제어 시스템"**을 구축하실 수 있습니다. 건승을 빕니다!