Introduction

The **ER-4U** robot was originally produced by **Eshed Robotics**, later renamed to **Intelitek Inc.**
The original Intelitek educational system was delivered as a complete mini-factory, including:

- CNC machines (BenchTurn, BenchMill)
- ER-series robots (ER-9, ER-14, ER-4U)
- A conveyor with multiple stations and storage solution

For this repository, the focus is on the **ER-4U robot** and its **peripherals**.
In my case, the peripherals include:

- A **small "gray" conveyor belt** (different from the larger storage conveyor)
- A **small rotation table** (not to be confused with the large automated storage carousel)

Purpose of This CLI

The goal of this project is to provide a **minimal, modern environment (2024)** example of how to use the **Intelitek usbc.dll API**.

This CLI is not intended as production software but as an **educational tool** for:

- Learning how to interact with the ER-4U via the USB controller
- Experimenting with most of the API functions
- Testing motion, I/O, and callback functionality
- Preparing for higher-level integrations (e.g., JSON over sockets, network control)

Development Notes

1. **Understand the ScorBase software**
   Before working with the raw DLL API, it is critical to familiarize yourself with the original **ScorBase** application.
   Exploring its functionality and options provides context for how the low-level API is structured.

2. **Limited official documentation**
   The usbc.dll library does not come with a formal API reference. However, two useful resources exist:

   - An **MFC sample project** provided by Intelitek (manualMovement, for Visual Studio 2008, Windows 7)

   - A **notes document** from an Intelitek customer describing API usage

3. **Reverse engineering with API Monitor**
   Even with the header files (usbc.h) and known capabilities, the correct **sequence of API calls** is not always obvious.
   To solve this, I used **API Monitor v2 alpha-r13**, which allowed me to trace function calls made by the original Intelitek software into usbc.dll.

A Note on Controllers

The usbc.dll is designed for the **Intelitek USB Controller**, used with the ER-4U.
There is also an **Intelitek USB-Pro Controller** (used with ER-9 and ER-14).

Although untested here, evidence suggests the APIs are nearly identical, and this CLI code may work with ER-9/ER-14 if compiled against **usbb.dll** instead.

Build & Run

Requirements

- **Windows 11** (tested)
    - Windows 10 should also work, but not verified
- **Visual Studio Code 2025** (as the editor)
- **Microsoft Visual Studio 2022 Build Tools** (for compilation)
- **Intelitek SCORBASE 2020** software
    - Must be installed and configured for your robot
    - Verify SCORBASE is able to connect and operate the robot before using this CLI
- **Intelitek usbc.dll and usbc.lib**
    - ⚠️ usbc.dll is **32-bit only**, so your project must be built for **Win32** (x86), not x64
- Hardware:
    - Intelitek **ER-4U robot**
    - USB Controller (connected and powered on)
    - Optional peripherals (small conveyor belt, small rotation table)

Required Files

These files come with the **SCORBASE 2020** installation, usually located in:

- C:\Intelitek\SCORBASE\BIN

For the CLI project to run, copy the following files into your project directory (override if needed):

- ER4CONF.INI – **most important file**, contains system configuration (gripper type, peripherals, etc.)
- Scbs.eng – SCORBASE language resource
- Scbs.ini – SCORBASE configuration file
- usbc.dll – the main USB controller API DLL (32-bit)
- usbc.ini – USB controller settings
- usbc.lib – import library needed for compilation
- Par\ folder – contains axis configuration (do not edit manually)
    - If changes are required, use the SCORBASE application to configure axes

⚠️ **Do not edit ER4CONF.INI or the Par folder manually.** Always configure through SCORBASE and then copy updated files if needed.

    -

Build Instructions

1. Open a **Developer Command Prompt for VS 2022**.
   (Make sure you choose the **x86 version**, since the DLL is 32-bit.)

2. Navigate to the project folder containing main.cpp, RobotController.cpp, and CallBack.cpp.

Compile the project:

cl.exe /nologo /EHsc /Zi /MD main.cpp RobotController.cpp CallBack.cpp /Fe:er4u-cli.exe usbc.lib ws2_32.lib

3.
   - Ensure usbc.lib is in your library path.

   - usbc.dll must be available at runtime (either placed next to the executable or copied into C:\Windows\System32).

4. Run the CLI:

   er4u-cli.exe

First Run

Once all requirements are in place, the files are copied, and the tool is compiled

1. **Check that SCORBASE works first**
    - Start the official **SCORBASE 2020** application
    - Verify that you can **initialize the robot** and **jog it manually**
    - **Close SCORBASE** before running the CLI (only one program can talk to the USB controller at a time).
2. **Run the CLI**
    - In the terminal:
3. Er4u-cli.exe
    - You should see the interactive menu:

--- Robot CLI Menu ---
1. Initialize robot
2. Check Emergency Status
3. Check Teach Mode
...
0. Exit

4. Test basic functionality

Initialize Robot (Menu 1)

- **Press 1 → Initialize robot**
- Expected:
    - The power light on the controller changes from **orange → green** (motors still OFF)
    - Console: [CLI] Robot initialized successfully.

Note: Initialization may take several seconds. The API returning “OK” is not enough; wait for the callback:

[InitEnd] Initialization finished

Mode: 1 Controller type: 41 Axes: 8

-

Enable Control (Menu 15)

- **Press 15 → Control ON/OFF**
- Expected:
    - The **motors light** on the controller turns green
    - Console: [RobotController] Control ON succeeded

⚠️ Note: Calling **Home** (Menu 18) also turns control ON automatically.
However, after a failed motion (e.g., hitting an obstacle), you will need to call **Control ON** manually to re-enable the drivers.

Home Robot (Menu 18)

- **Press 18 → Home Robot**
- Expected:
    - The robot homes all axes
    - Console: [CLI] Homing Robot command sent.

> 💡 Cartesian (XYZ) moves are usually available **only after homing**, but joint moves can work even before.
> Homing **robot** and **peripherals** should be done separately.
> The API offers “home all axes”, but this may fail if some peripherals are not connected.

Open/Close Gripper (Menu 11 & 12)

- **Press 11 → Open Gripper**
- **Press 12 → Close Gripper**
- Expected: Gripper moves accordingly

Get Current Position (Menu 24)

- **Press 24 → Get Current Position**

Expected: Console shows encoder, joint, and XYZ values for each axis, e.g.:

Axis 0 enc=0 jnt=0 xyz=169030

Axis 1 enc=0 jnt=-120278 xyz=0

Axis 2 enc=0 jnt=95021 xyz=504328

...

CLI Command Reference

The CLI is designed as a **1:1 mapping** to the usbc.dll API.

- Menu numbers match the feature groups.
- Console output shows exactly which call was made and the returned result.
- Minimal logic is added — the goal is to stay close to the raw API for easier debugging and testing.

The only exception is **Teach point (22)**, which has extended logic for user-friendly input.

🔧 Configuration Functions

- **1. Initialize Robot**
  → Calls: Initialization() with callbacks InitEnd and ErrorMessage.
  Prepares the controller. Must be called before anything else.
  Internally it initializes two callbacks. The output of them (and any other callbacks) is printed to the terminal

- **2. Check Emergency Status**
  → Calls: IsEmergency()
  Returns whether the emergency stop is active.
  The emergency button is located on the controller itself, and another one on the teach pendant (if used).
  In scorebase, this is checked a few times per sec. It's checks the Mode 2-3 times, then emergency, and again Mode and so on

- **3. Check Teach Mode**
  → Calls: IsTeachMode()
  Indicates if Teach Pendant mode is enabled.
  (physically, on the pendant there is an auto/teach switch)

- **4. Check Online OK**
  → Calls: IsOnLineOk()
  Verifies connection with the controller.

- **5. Get Configuration**
  → Calls: GetConfig()
  Dumps configuration from ER4CONF.INI (controller type, axes, gripper, digital/analog IO

counts, software version, etc.).

- **6. Get Mode**
  → Calls: GetMode()
  Returns the controller's operating mode.

  Personally i did not get the idea of this api.
  On the Scorebase it's called few times a second. The result may be 0 - offline, 1 - online, 2 - simulator mode.
  From my experience with the robot. It's 0 before init, and 1 after. No need to call it each time

---

🤖 Motion Functions

- **7. Enter Manual Mode**
  → Calls: EnterManual()
  Switches to manual movement (e.g. joint jog or linear jog).
  0 means joints mode, so you can control each joint separately.
  Note that the "normal" speed for a robot is 50, but for the peripherals is 10. (Peripherals are axis 6,7,
  robot starts with 0 for base rotation)

- **8. Move Manual Axis**
  → Calls: MoveManual(axis, velocity)
  Jog an axis continuously until stopped.
  Depends on what you have selected on EnterManual().
  For the robot it is basically 5 axes. 0 to 4. 5 is the gripper motor.6 and 7 is peripherals. If you selected joints, so each number is an axle, if linear/cartesians so axis 0 means X, 1 is Y, then Z, Peach, and Role
  For the peripheral, it should be a distance in cm (for the belt) but have never checked.
  Note that positive and negative values of the speed is available, setting the direction, speed 0 is for stop

  Once you have entered a command, the robot will move for a few seconds. This is too much for one command.
  So you can override it by another command, or call CloseManual() that also will stop the movement and cancel the moveManual.

  If you reach the "end time" of the command/movement, you can not call another one. You have to call the EnterManual again

- **9. Close Manual Mode**
  → Calls: CloseManual()
  Exits manual jogging mode. Stops the motion

- **10. Move to a Taught Point**
  → Calls: MoveJoint() or MoveLinear() depending on mode.
  Moves to a defined point in a vector. Supports two vectors if needed.

  This one is a bit complex. See the appendix at the end of the document

- **11. Open Gripper**
  → Calls: OpenGripper()

- **12. Close Gripper**
  → Calls: CloseGripper()

- **13. Stop Motion**
  → Calls: Stop()
  Stops movement of all axes.

- **14. Move with Velocity**
  → Calls: Velocity(groupOrAxis, vel)
  Continuous velocity control.

  Note: this api is used to move the gray belt conveyor by Scorbase. Personally, I have failed to make it work. So i move the belt conveyor with moveManual

- **15. Control ON/OFF**
  → Calls: Control()
  Enables/disables power to all axes.

- **16. Watch Control**
  → Calls: WatchControl(callback)
  Subscribe to control state changes.

- **17. Stop Watching Control**
  → Calls: CloseWatchControl()

- **18. Home Robot**
  → Calls: Home('A', callback)
  Homes all robot axes. It's a wrapper to general Home API.
  To home only the robot call it with 'A'. to home peripherals call with 'B'. to home all, use '&'. But it may fail if there is no peripherals, of they are not set properly

- **19. Home Peripherals**
  → Calls: Home('B', callback)
  Homes peripheral axes (conveyor, rotary table, etc.).

📍 Position Functions

- **20. Define Vector**
  → Calls: DefineVector(group, name, dim)
  Creates a vector storage for points (must be done before teaching or moving).
  Groups:
    - A = SCORA (robot arm)
    - B = SCORB (peripherals)

  See the appendix by the end of the document

- **21. Here (store current position)**
  → Calls: Here(vectorName, point, ABS_JOINT)
  Saves current axis positions into a vector/point.

  See the appendix by the end of the document

- **22. Teach Point (manual values)**
  → Calls: Teach(vectorName, point, coords[], dim, pointType)
  Details:
    - CLI prompts the user for coordinates one by one.
    - Typical dim = 5 (X, Y, Z, P, R).
    - pointType = ABS_XYZ_A by default (absolute Cartesian).
    - Use this when you want to manually insert coordinates instead of using the robot's current position.

      See the appendix by the end of the document

- **23. Get Point Info**
  → Calls: GetPointInfo(vectorName, point, enc, joint, xyz, type)
  Dumps stored encoder, joint, and XYZ values for a point.

- **24. Get Current Position**
  → Calls: GetCurrentPosition(enc, joint, xyz)
  Reads live encoder/joint/XYZ values for all axes.

📊 Monitoring Functions

- **25. Start ShowXYZ**
  → Calls: ShowXYZ(callback)
  Streams live XYZ data via callback.

- **26. Stop ShowXYZ**
  → Calls: CloseXYZ()

- **27. Start ShowTorque**
  → Calls: ShowTorque(axis, callback)
  Streams live torque data for one axis.

  Note that the accuracy is not perfect, and the units' are not known (to me). From my test, if you push/pull the robot, you can see a significant change in the values, and note the it not always returns to the same when you release the force on the arm

- **28. Stop ShowTorque**
  → Calls: CloseTorque()

- **29. Watch Motion**
  → Calls: WatchMotion(MotionEnd, MotionStart)
  Triggers callback events when motion starts/ends.

- **30. Stop Watching Motion**
  → Calls: WatchMotion(NULL)

🔌 Input / Output Functions

- **31. Watch Digital Inputs**
  → Calls: WatchDigitalInp(callback)

- **32. Stop Watching Digital Inputs**
  → Calls: CloseWatchDigitalInp()

- **33. Get Digital Inputs**
  → Calls: GetDigitalInputs(&bitmap)
  Reads a 32-bit input bitmap.

  Note: this function works, only if WatchDigitalInputs activated

- **34. Set Digital Output**
  → Calls: SetDigitalOutput(number, state)
  Turns a single digital output ON/OFF.

- **35. Get Digital Outputs**
  → Calls: GetDigitalOutputs(&bitmap)
  Reads the full digital output bitmap.

  There is also an option for Analog read/write,
  Not implemented in this example.

  Note about power. The digital output power limit for outputs 1-4 is 1A, for other are 0.5A
  For the analog is 20mA max. Check the scorbase documentation for exact spec and connection options.

Appendix about move command, and it's belongings

For just moving the robot, you can use the MoveManual commands. It will move each axel, of by XYZ depends on the mode. But most of the robotis system prefer a pre-defined points, and a command in style goto(pointX).

To achieve this functionality, we need an array to store the points. When you use scorabese, the arrays is defined in the background, but here we should do it manually.

Actually, there is two arrays needed. One for the robot/arm coordinates, and another one for the peripherals.

So, before defining a point in the working area, you should call defineVector. You can check the logic in the code, but basically, it ask you for the array size (the scorbase default is 1000), and will define two arrays, named SCORA and SCORB. (note: i've tried to call them v1 and v2, but it does not work. Maybe only capitals, or no numbers allowed)

Once you have defined (and initiated) the arrays, you can teach a point. The easiest way, is to bring the robot there, and call Here api. In the usbc API you should provide a vector name, type, and point number. I've wrapped it with a logic that you need to provide only the point number. The location will be saved both for SCORA and SCORB, at the same index.

You can get the data by calling getPointInfo. Here you should manually set the vector name, and the index. So you can check what vector contains what data.

And now we got to the move call.
First, there is two exposed typs of movement. The joint, and the linear.
Joint means that the robot will do optimal (from his perspective) path to the point
Linear means it will move the tip of the griper in a strait line, that may be not so efficient.

At scorebase there is also a moveCircular, where you can move to point C via point B
At the api there is also a few more move calls. Never used them.
It's told that the moveTorque os the most dangerous, as it disable torque limit at the controller.
For my teaching experience, i think that move joint and linear is enough for most of the cases.

You can use Teach API to set a point with known coordinates, without moving the robot where.
The default type of the point is ABS_XYZ_A that is absolute cartesian point.
Note that in scorebase the point are 1/100 of a cm, but in code they are 1/1000.

Probably , relative points can be defined, but i prefer to define them in code on a client side, rather than on the controller.