INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2026, VOL. 72, NO. 3, PP. 1-6
Manuscript received June 01,2026; revised June, 2026. doi: 10.24425/ijet.2026.157951

A shared register specification as the source of complete Modbus RTU device support

Emilian Świtalski, and Krzysztof Górecki

***Abstract*—Since 1979, Modbus RTU has remained one of the most popular industrial communication protocols, yet it carries no information about the meaning of the registers it transmits. As a result, the same description (type, unit, range, factory value) appears in parallel across the device firmware, the client software, the user interface, and the documentation. This article presents an approach in which a single CSV text file serves as a shared source of truth for all layers that handle a device. We describe the concept of a register descriptor together with its implementation in the open-source application Modra, which runs both as a desktop program and as an HTTP server. The application was verified on a case study controlling the inverter of a permanent-magnet synchronous motor (PMSM).**

***Keywords*—Modbus RTU; RS-485; Register Descriptor; Automatic User-interface Generation; Single Source of Truth; Industrial Automation; Open Source; PMSM Motor; Inverter; OpenCPLC; CSV**

I. Introduction

MODBUS RTU is one of the most widely used industrial communication protocols [1]. Developed in 1979 and later formally adopted into the IEC 61158 family of standards[2], it is still the first choice when designing inverters, meters, sensors, and PLCs. The reason is pragmatic: the implementation is cheap, the RS-485 bus is two-wire and simple, and on the client side the protocol is supported almost everywhere.

Table I summarizes the position of common industrial communication protocols. The percentages are approximate, based on the HMS Industrial Network Market Shares 2024 report [3] supplemented with estimates for related markets [4]; they are meant to illustrate the scale and distribution of deployments rather than to serve as precise measurements.

The protocol transmits only 16-bit registers, with no information about their meaning. The type, unit, range, and description of each register exist only in the manufacturer's documentation. The same description appears in parallel in the device firmware, in the communication layer of the client software, in the user interface, and in the technical documentation. Four copies mean four places where inconsistency can arise.

Other industrial protocols solve this problem in two ways. CANopen, PROFIBUS, and IO-Link store metadata in external description files (EDS, GSD, IODD). BACnet[5] and OPC UA embed them directly in the protocol, so they are accessible through the same mechanism as the values themselves. Modbus has neither of these (Table II).

TABLE I
POSITION OF COMMON INDUSTRIAL COMMUNICATION PROTOCOLS (APPROXIMATE VALUES)

Year	Family	Comple-xity	Characte-ristics	In devices	Market share
1979	Modbus RTU/TCP	🟢 trivial	ubiquitous, every OEM, open	~95%	~7%
1986	HART	🟡 medium	dominant in process measure-ment	~25%	~3%
1989	PROFIBUS /PROFINE T	⚫ high	leader of the Siemens ecosystem	~35%	~30%
1995	CANopen	🟡 medium	drives, automotive	~15%	~2%
1995	BACnet	⚫ high	dominant in building automation	~25%	~7%
1996	EtherNet/IP (CIP)	⚫ high	dominant in the USA (Rockwell)	~30%	~20%
2003	EtherCAT	⚫ high	high-performan ce motion systems	~20%	~15%

TABLE II
COMPARISON OF COMMON INDUSTRIAL COMMUNICATION PROTOCOLS WITH RESPECT TO REGISTER DESCRIPTION

Protocoł	Metadata location	Description standard	Addressing	Implementa-tion barrier
Modbus RTU	none	none	16-bit register	🟢 very low
CANopen	external	EDS	index + sub-index	🟡 low
PROFIB US	external	GSD	slot + index	⚫ high
IO-Link	external	IODD	port + index	🟠 medium
BACnet	in protocol	EDE	obiect + property	⚫ high
OPC UA	in protocol	NodeSet XML	NodeId, hierarchy	⚫ very high

Publication prepared within the project financed from the state budget under the "Wektory Nauki" programme of the Minister of Science and Higher Education, project no. WNK/SP/0311/2025/01. Project funding: PLN 276,232.00. Total project value: PLN 635,552.00.

Emilian Świtalski and Krzysztof Górecki are with Gdynia Maritime University, Department of Power Electronics, Gdynia, Poland (e-mail: e.switalski@we.umg.edu.pl, k.gorecki@we.umg.edu.pl).

This paper presents a client-side solution to this problem: a register descriptor format stored in a text file, an implementation of the open-source application Modra that generates device handling directly from the description, and a verification of the concept on a PMSM inverter. The following sections cover: the Modbus RTU protocol, the CSV descriptor, the architecture of the Modra application, tools for working with Modbus, and an example of the application in use.

II. Protocol Modbus RTU

Modbus is an application-layer protocol that operates in a master-slave model with a single master and up to 247 slave devices [1]. Each frame contains the slave address, a function code Fn specifying the type of operation (reading or writing registers), the data, and a CRC-16 checksum (Fig. 1). The exchange follows a request-response pattern (Fig. 2). The RTU variant uses binary transmission, and the physical layer is usually RS-485, a multidrop bus with differential signaling. Modbus operates on four disjoint address spaces (coils, discrete inputs, input registers, holding registers), although in practice most manufacturers expose all variables as holding registers.

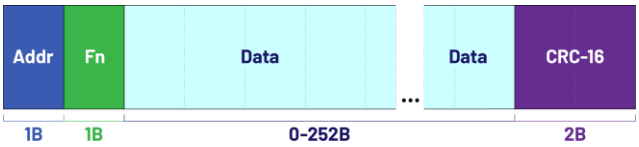

Fig.1. Structure of a Modbus RTU frame

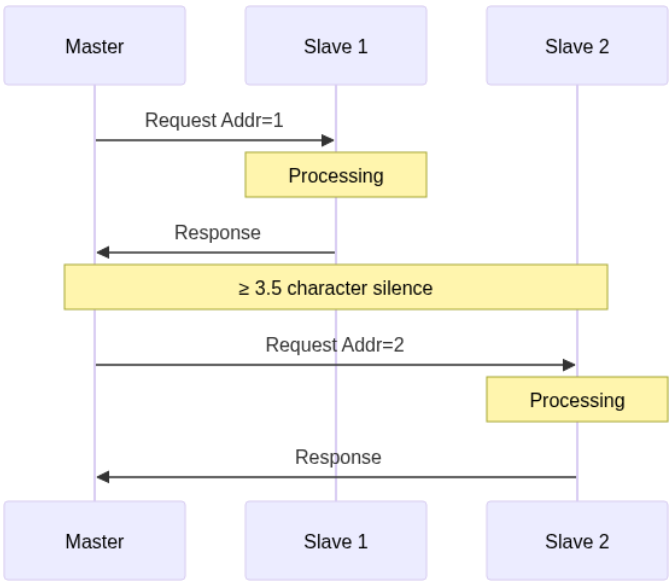

Fig. 2. Master-slave exchange in the Modbus RTU protocol

Modbus RTU is simple to implement and robust in operation, but it solves only the problem of data transport. The description of the registers remains outside the protocol. The same applies to the "no data" signal: a disconnected sensor or an inactive measurement channel returns a 16-bit word like any other register. For this case, manufacturers reserve boundary values outside the usable scale: 0xFFFF for unsigned types, 0x8000 for signed types, and NaN for floating-point types. This convention has been codified, among others, by the SunSpec standard in the photovoltaic industry [6], which goes a step further and imposes fixed register models for PV devices; it is, however, rigidly tied to that domain, whereas the proposed descriptor assumes no particular device type.

III. CSV descriptor

Each row describes one register, and each column describes one of its attributes. The format was chosen with three characteristics in mind:

- human readability and editability in a spreadsheet, together with a readable diff in version control,
- no compilation and no specialized editor,
- one file, one source of truth.

CSV was chosen over YAML and JSON because the descriptor is flat (a table of registers, not a nested structure), and the spreadsheet remains the everyday tool of the industrial integrator.

The idea of a single description as the source of truth for an entire control system is not new. A classic example is EPICS, used in particle accelerators, where this role is played by a record database [7]. Beyond industrial automation, the same pattern appears in specifications for networked devices: the W3C Web of Things Thing Description describes IoT devices together with their interactions and data schema in JSON-LD[8], AutomationML (IEC 62714) standardizes the exchange of engineering models in XML [9], and the OPC UA NodeSet provides a hierarchical representation of the address space, also in XML [10]. All three assume a nested structure, a formal schema, and dedicated editors. The proposed approach goes in the opposite direction: a single flat text file with no schema and no special editor, matched to the equally flat Modbus address space. The proposed descriptor has 12 columns (Table III).

Table III
Columns of the CSV descriptor

Column	Description	Example
id	register address (decimal)	11
hex	hexadecimal address (optional)	0x0B
name	name in the format Group:Field	Ctrl:Setpoint
rws	access mode (R, W, RW, RWs)	RW
type	data type	uint
unit	measurement unit or enum labels	rpm
scale	value multiplier	10
min	minimum value	0
max	maximum value	10000
desc	text description	Setpoint sterownika
rule	dynamic rule (optional)	switch=Ctrl:Mode
default	factory value (optional)	1500

The rws field takes four values:

- R: read-only (measurements, status).
- W: write-only (triggers, resets).
- RW: read-write, volatile (setpoints, operating modes).
- RWs: read-write, persistent (configuration).

The rws field also acts as a write barrier shared across all layers: the frontend hides editing controls for R registers, the backend rejects writes at the API level, and the firmware generates register addresses and attributes from the same file - there is no layer in which a read-only register could be overwritten by accident.

The type field covers numeric types (uint, int, float), boolean (bool), enumerated (enum), and hexadecimal (hex); the ? prefix (e.g. ?int) declares a register that may return the "no data" sentinel, which the client interprets as a gap rather than an out-of-scale value. The default field holds the factory value and matters for RWs registers, because the same value appears in the device firmware, the production tool, the reset procedure, and the user documentation.

The descriptor also supports rules (rule) for custom relationships between registers, linking them into pairs or larger groups with a shared meaning. Two such rules are currently defined. One ties a register's scaling to the state of another register, a typical example being a drive setpoint that changes its unit depending on the control mode (rpm, %, Hz). The other combines a pair of registers into a single 32-bit value, either integer or floating-point in IEEE 754 format. The mechanism is open; adding further relationship types requires no changes to the descriptor format.

The same principle extends to the default column and allows different factory values to be defined for hardware variants of a product. An example is drives with motors of three rated powers, for which the factory maximum speed differs depending on the selected variant.

The regs.csv file acts as a shared contract between the layers that handle a device, from which complete handling can be generated automatically at five levels:

- Backend: encoding and decoding of values, range validation, the polling loop, write auditing.
- Frontend: UI controls (enum → buttons, bool → toggle, numeric → field with validation), units, grouping, charts.
- Database: an SQLite table schema with a column for each register.
- Firmware: C headers with addresses, enums, range macros, and factory-value initializers.
- Documentation: register tables with descriptions, units, ranges, and factory values in Markdown and LaTeX formats.

The firmware level is illustrated by the following fragment of a generated header, with an enum of the address space and a factory-value initializer.

```
typedef enum {
REG_Ctrl_Mode, ...,
REG_Speed_Max, ...
} REG_t;

uint16_t Default[REG_COUNT] = { ...
[REG_Ctrl_Mode] = STATE_Mode_Disable,
[REG_Speed_Max] = 3700,
...};
```

The idea of automatically generating Modbus connectors from a declarative description has already been raised in the literature [11], but the proposed approaches relied on heavy component platforms. The Modra application, which demonstrates the concept described here, implements three of these levels in a minimalist way: backend, frontend, and database schema. The generator of C headers for firmware and of documentation tables consists of separate tools that work on the same regs.csv file within a broader project. A single CSV file is enough to launch a complete client with a user interface.

IV. Architecture of the Modra Application

The Modra application was developed by one of the co-authors of this paper as a tool for managing Modbus RTU devices described with a common register descriptor. The application's source code and ready-made executable files are available in the repository [16].

The regs.csv file is the source of truth for the entire system (Fig. 3). The backend parses it at startup and exposes it to the frontend through an API call, so that both layers of the application work on the same description. Communication with the device takes place over RS-485, the register history is stored in an SQLite database, and the user interface is generated in the browser directly from the descriptor fetched from the backend.

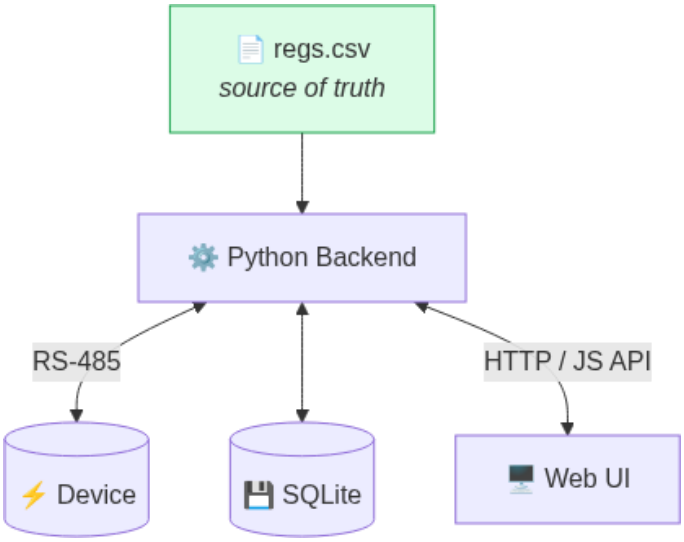

Fig.3. Architecture of the Modra application

The application runs in two modes that share a common code base:

- Desktop (PyWebview): a single executable containing the backend, the frontend, and an embedded browser engine, running locally.
- HTTP server: the backend exposes a REST/JSON interface, the frontend is served as static files, and access is through any browser, including a distributed setup with the backend next to the device and the frontend on the operator's machine.

Synchronization between the backend and frontend occurs in two stages. The descriptor is retrieved once at session startup as a static description of the register space, and the current values are delivered via an asynchronous polling loop with a configurable period. This loop simultaneously determines the bus query rate and the interface refresh rate, ensuring that both layers remain synchronized without a separate mechanism between the network layer and the presentation layer. Operator writes leave the client as a single transaction, each block of

which is verified by an immediate readback from the device; any write denial on the firmware side is revealed immediately, not only in the next polling iteration.

In industrial automation systems, Modra sits between lightweight Modbus client libraries and heavy SCADA platforms. It requires neither the definition of process variables nor a separate database. Its configuration fits in a single CSV file, which makes the application well suited as a laboratory and service tool for prototyping devices. The application's source code and ready-to-use executables are available at https://github.com/Xaeian/Modra.

V. User Interface

Modra's interface is built entirely at application startup, directly from the regs.csv file. The presentation layer contains no code specific to a particular device; it is only an interpreter of the descriptor.

The core of the interface is the register grid, in which each row of the descriptor corresponds to one row in the grid (Fig. 4). The control type is chosen based on the type column: numeric registers get a numeric field with range validation based on min/max, enum registers are rendered as mutually exclusive buttons labeled from the unit column, and bool registers as HIGH/LOW button pairs. The unit suffix, the description, and the access-mode badge come from the remaining columns of the descriptor (Table IV).

TABLE IV
MAPPING OF REGISTER-GRID ROW ELEMENTS TO DESCRIPTOR COLUMNS

Row element	Source in the descriptor
Address 1	id
Ctrl:Setpoint	name, the Ctrl: prefix is used for grouping
Field 1000	type, validation based on min/max
rpm	active unit value
▟ ⊘ ⚙	row icons: chart, ignore, slider
Badge RW	rws, color reflects the access mode

The toolbar (Fig. 5) combines control of the serial connection with control of the communication: port selection, slave address, connecting, manual reads, and sending changed values. The expandable settings panel contains link parameters (baud rate, parity, stop bits, timeout), the polling interval, the retry budget, address-range scanning, and import/export of current values to INI or CSV files. The link parameters come from a separate serial.ini file, since they do not belong to the register specification of any particular device.

The chart panel (Fig. 6) handles the historical aspect of the application. R registers with a numeric type can be added to a chart by clicking the ▟ icon in the register row. The mechanism groups series by shared unit and scale, so that physically comparable quantities end up on the same panel. Changing the time range triggers loading history from the data.db database, which makes it possible to observe traces from before the application was started as well.

Fig.4. A single row of the register grid in the Modra application

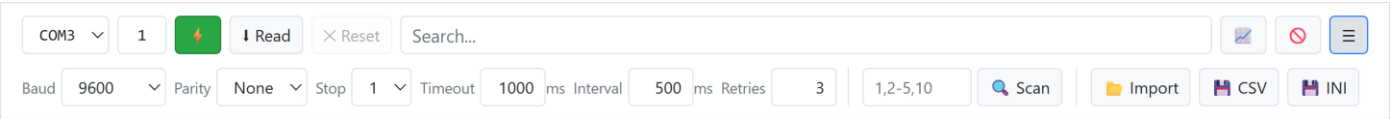

Fig. 5. The Modra toolbar with the settings panel expanded.

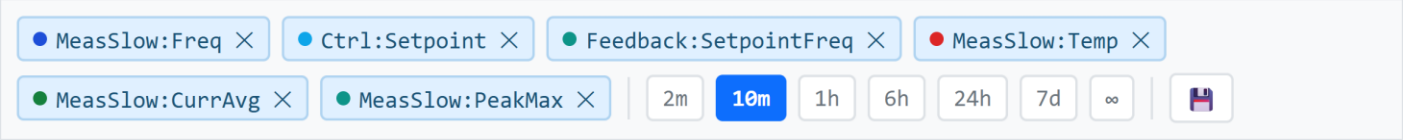

Fig. 6. The Modra chart panel with active measurement series.

Beyond the control chosen from the type column, a row encodes what the descriptor does not, namely the state of the exchange with the device. A change not yet sent tints the row, a value outside its range outlines the field in red, and a register taken off polling has its name struck through; all of this clears once the device acknowledges the write. Edits are kept apart from the incoming readings, so they are not lost to a refresh and leave the client as one commit rather than a sequence of independent writes.

A register can be withheld from editing in two ways. The rws barrier removes the control on a read-only register, so a write cannot be made there by accident. Ignoring takes a register off polling and out of the grid, tidying the derived state by dropping any pending edit and removing it from the charts; a separate toggle shows the skipped registers struck through, so their set stays auditable.

What to observe is the operator's choice: clicking the chart icon adds a numeric register as a variable tracked over time, back-filled from history at once, and the chosen set is saved to the view file and survives a restart. The link parameters, that is baud rate, parity, stop bits, timeout, interval and retries, together with the scan over addresses 1 to 247, describe the session rather than the device, so they live in serial.ini and not in the descriptor; changing a wire-level field invalidates the port and forces a reconnect.

The configuration held in the RWs registers can be preserved off the device. Export writes their state to a .csv or .ini file with units, so it can sit beside the descriptor in version control. Import

does not write to the bus directly but replays the values into the edit buffer, where the operator reviews them before sending; a saved configuration therefore returns deliberately, not blindly.

VI. Tools for Working with Modbus

Popular tools for working with Modbus (Table V):

- libmodbus, pymodbus: libraries that provide the protocol layer, but handling a device requires writing one's own application.
- ModbusPoll, Radzio Modbus Master: desktop GUIs with manual register mapping, saved in a binary project file.
- ScadaBR, Mango Automation: SCADA platforms with a process database, dashboards, and scripts; the interface is designed separately for each dashboard.
- Node-RED with a Modbus module[12]: a flow-based environment in which communication is modeled as a graph of nodes, and the dashboard is composed manually.
- OpenPLC Editor[13]: an IDE for programming PLCs in accordance with IEC 61131-3, in which Modbus serves as the I/O layer for a ladder program.

Table V
Comparison of common tools for handling Modbus devices

Tool	Per-device configuration	Entry barrier	Customization barrier	History
libmodbus, pymodbus	custom code	● high	● low	● custom implementation
ModbusPoll, Radzio	click-based mapping (binary)	● low	● impossible (closed source)	● limited
ScadaBR, Mango	configuration + scripts	● high	● moderate	● full
Node-RED + Modbus	flow + dashboard	● moderate	● moderate	● via node
OpenPLC Editor	ladder program	● high	● low	● limited
Modra	single CSV file	● low	● high	● full

Among the tools listed, Modra is the only one that combines a low entry barrier with a declarative register description in a CSV text file that can be both edited in a spreadsheet and tracked in the GIT version control system. The price is a high customization barrier, because changing the frontend requires familiarity with a custom, in-house solution. For the other tools the trade-off looks different: libraries offer full freedom at the cost of writing an application from scratch, commercial GUIs are easy at first but closed to change, and SCADA platforms only make sense for multi-device installations.

VII. Practical Example: A PMSM Motor

The permanent-magnet synchronous motor with an integrated three-phase inverter is a product of Ectra Group sp. z o.o., for which the authors of this article developed firmware based on the open OpenCPLC platform [14, 15]: https://github.com/OpenCPLC. The Modra application is a general-purpose tool: a new CSV descriptor means support for another device with no changes to the client code, and the PMSM inverter is here only a practical verification of the concept. The test setup amounts to a motor connected to a laptop through a USB↔RS-485 converter (Fig. 7, Fig. 8); from the application's point of view this is an ordinary COM serial port carrying Modbus RTU.

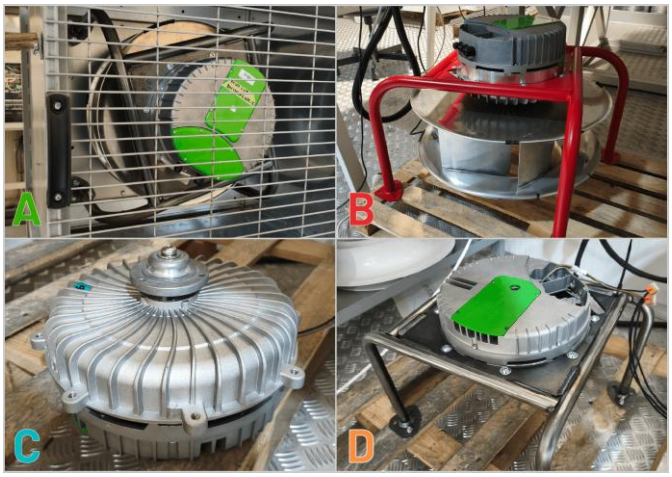

Fig. 7. PMSM motors (A, B, C, D)

The full inverter descriptor comprises more than three hundred registers conforming to the structure described above. The fragment below shows a few representative rows: the operating mode as an enum, a setpoint with a switch rule that changes its unit depending on the mode, the maximum speed with three factory values for the power variants, and measurement registers of type ?uint/?int that handle the "no data" sentinel.

```
id,hex,name,rws,type,unit,scale,min,max,desc,rule,default
0,0x00,Ctrl:Mode,RWs,enum,0:off 1:ai 2:% 3:rpm 4:Hz,-,-,-
,Motor control mode,,off
1,0x01,Ctrl:Setpoint,RW,rule,rpm/%/Hz,1/100/100,0/0/0,500
0/100/300,Setpoint; interpretation depends on
Ctrl:Mode,switch=Ctrl:Mode,
9,0x09,Speed:Max,RWs,uint,rpm,1,0,10000,Maximum motor
speed,,3700/2600/1500
33,0x21,MeasSlow:Freq,R,?uint,Hz,100,,,Output frequency,,
37,0x25,MeasSlow:CurrAvg,R,?uint,A,1000,,,Average RMS
current of three phases,,
41,0x29,MeasSlow:PeakMax,R,?uint,A,1000,,,Max peak-to-
peak amplitude of three phases,,
42,0x2A,MeasSlow:Temp,R,?int,°C,100,,,Power stage
temperature,,
```

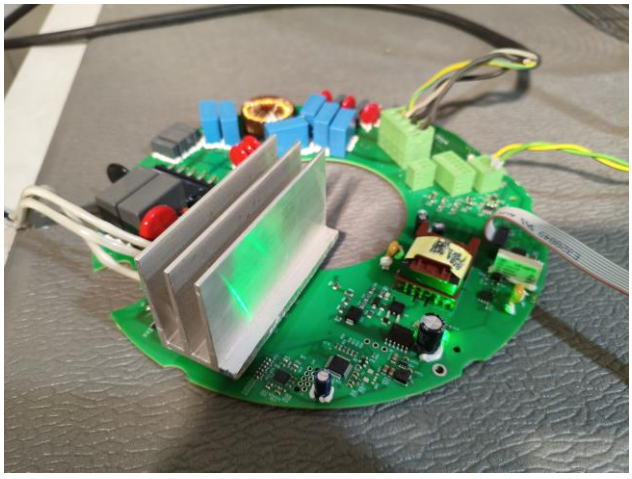

Fig. 8. Integrated off-motor inverter, PMSM

During the iterative development of the inverter firmware, the register layout changed between successive revisions: addresses, the order of groups, and added or removed factory values for the power variants. A single update to regs.csv then propagated to the frontend, the backend, and the firmware's C headers, keeping all layers consistent without manual synchronization. The value of the descriptor showed itself more clearly here than for a single device with a stable register map.

Test scenario: in Hz mode the setpoint is set to 100, which corresponds to a synchronous speed of 2000 rpm, and is then brought back to zero. In real time, Modra plots the measured frequency together with the setpoint, the phase currents (average and peak), and the power-stage temperature. Figure 9 shows how the frequency rises to the setpoint, with currents and temperature rising along with it, after which all quantities fall during the coast-down phase. The entire interface was generated directly from the descriptor, with no presentation-layer code added.

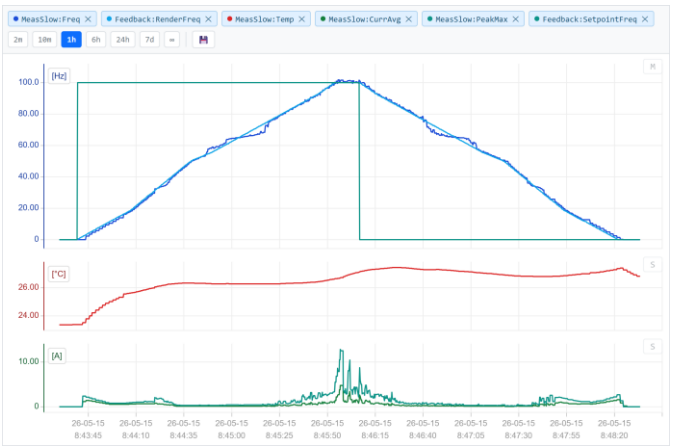

Fig. 9. Screenshot of Modra during a spin-up and coast-down cycle of the PMSM motor

VIII. Conclusions

Modbus RTU has survived 47 years precisely because it is simple and imposes no semantics. Other industrial protocols solved the problem of register description either with external files or with metadata built into the protocol. Modbus adopted neither. This is perhaps because the newer protocols had already solved the problem on their own. Modbus, however, remains the most popular industrial communication protocol, so filling this gap on its side has practical value. Any solution must therefore come from the outside, on the client side. The approach presented here fills this gap after nearly half a century.

The value of this approach does not lie in technology. The CSV format and the Modbus RTU protocol are essentially trivial. The value lies in the organization of work: when the firmware developer, the system integrator, and the person writing the documentation all work on the same file, synchronization follows from the design of the solution itself.

The full potential of this approach emerges when the descriptor file is shipped by the manufacturer together with the device, analogously to EDS files in CANopen, GSD in PROFIBUS, or IODD in IO-Link (Table II). Since the device firmware already contains this data as literals and definitions, generating the descriptor on the manufacturer's side is a natural step, and instead of retyping register tables from the documentation, the integrator simply points to a ready-made CSV file.

Modra is a development tool and a proof of concept, not a solution for building large communication networks. The concept finds practical confirmation in the example described above, controlling a PMSM inverter, where the same descriptor serves both the Modra application and the device firmware. An open question is whether the same pattern of a declarative description as the source of truth can be applied to simpler industrial protocols: raw CAN, custom UART protocols, or the I²C and SPI buses.

References

[1] Modbus Organization, MODBUS Application Protocol Specification V1.1b3, Modbus Organization Inc., Hopkinton, MA, USA, April 2012

[2] M. Felser, T. Sauter, The fieldbus war: history or short break between battles?, Proc. 4th IEEE Int. Workshop on Factory Communication Systems (WFCS), Västerås, 2002, pp. 73–80, doi:10.1109/WFCS.2002.1159702

[3] HMS Networks AB, Industrial Network Market Shares 2024, Halmstad, Sweden, 2024

[4] BSRIA Ltd., Market Penetration of Communications Protocols 2018-2027, Update 2024, BACnet International, Atlanta, GA, USA, 2024

[5] ANSI/ASHRAE Standard 135-2024 / ISO 16484-5:2022, BACnet: A Data Communication Protocol for Building Automation and Control Networks, ASHRAE, Atlanta, 2024

[6] SunSpec Alliance, SunSpec Device Information Model Specification, V1.2.1, SunSpec Alliance, January 2025

[7] L.R. Dalesio et al., The Experimental Physics and Industrial Control System Architecture: Past, Present, and Future, Nuclear Instruments and Methods in Physics Research A, vol. 352, no. 1–2, 1994, pp. 179–184, doi:10.1016/0168-9002(94)91493-1

[8] W3C, Web of Things (WoT) Thing Description 1.1, W3C Recommendation, December 2023

[9] IEC 62714-1:2018, Engineering data exchange format for use in industrial automation systems engineering - Automation Markup Language - Part 1: Architecture and general requirements, IEC, Geneva, 2018

[10] IEC 62541-3:2020, OPC Unified Architecture - Part 3: Address Space Model, IEC, Geneva, 2020

[11] C. Nikolajew, H. Eichelberger, Industry 4.0 Connectors: A Performance Experiment with Modbus/TCP, arXiv:2410.15813, October 2024, doi:10.48550/arXiv.2410.15813

[12] M. Tabaa, B. Chouri, S. Saadaoui, K. Alami, Industrial Communication based on Modbus and Node-RED, Procedia Computer Science, vol. 130, 2018, pp. 583–588, doi:10.1016/j.procs.2018.04.107

[13] T. Alves, T.H. Morris, OpenPLC: An IEC 61131-3 compliant open source industrial controller for cyber security research, Computers & Security, vol. 78, 2018, pp. 364–379, doi:10.1016/j.cose.2018.07.007

[14] E. Świtalski, K. Górecki, Otwarte i warstwowe projektowanie sterowników PLC na przykładzie sterownika do zastosowań edukacyjnych, Przegląd Elektrotechniczny, vol. 100, no. 10, 2024, pp. 285–288, doi:10.15199/48.2024.10.61

[15] E. Świtalski, K. Górecki, Evolution of Programmable Boards as a Response to Market Changes and Technological Paradigm Developments, Electronics, vol. 14, no. 24, 2025, art. 4810, doi:10.3390/electronics14244810

[16] E. Świtalski, Modra - open-source Modbus RTU register descriptor client, GitHub repository, https://github.com/Xaeian/Modra, accesssed: 21.05.2026.