ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV039

A RELIABLE MONITORING AND CONTROL SYSTEM FOR VACUUM SURFACE TREATMENTS

J. Tagg, E. Bez, M. Himmerlich, A. K. Reascos Portilla, CERN, Geneva, Switzerland

Abstract

Secondary electron yield (SEY) of beam-screens in the LHC puts limits on the performance of the accelerator. To ramp up the luminosity for the HiLumi LHC project, the vacuum surface coatings team are coming up with ways to treat the surfaces to control the electron cloud and bring the SEY down to acceptable levels. These treatments can take days to weeks and need to work reliably to be sure the surfaces are not damaged. An embedded control and monitoring system based on a CompactRIO is being developed to run these processes in a reliable way [1].

This paper describes the techniques used to create a LabVIEW-based real-time embedded system that is reliable as well as easy to read and modify. We will show how simpler approaches can in some situations yield better solutions.

PROJECT AND BACKGROUND

The objective of the LESS (Laser Engineered Surface Structures) project is the commissioning of an in-situ laser surface treatment conceived to mitigate electron clouds in the Large Hadron Collider (LHC) at CERN. Secondary electrons are multiplied when they interact with the vacuum chamber walls of the accelerator and consequently form electron clouds that can negatively affect its performance.

The secondary electron emission of a surface can be reduced by surface roughening. In this project, pulsed laser processing is applied to generate micro and nanostructures on the inner vacuum chamber surface that surrounds the proton beam. In this way, secondary electrons are captured by the surface geometry. The resulting structures and the performance of the surface strongly depend on the processing parameters, such as the laser power, the scanning speed, and the line distance, as well as on the scanning pattern [2].

The final treatment must be applied in-situ in the already existing accelerator and the system must be capable of treating tens of meters of vacuum pipe autonomously. The dedicated setup to perform this is composed of a picosecond pulsed laser source and a Beam Delivery System (BDS) that shapes and couples the laser beam into an optical fiber, which guides the laser light through an inchworm robot where the beam is decoupled through a rotating nozzle (see figure 1). The translational movements of the robot are driven by a pneumatic clamping system.

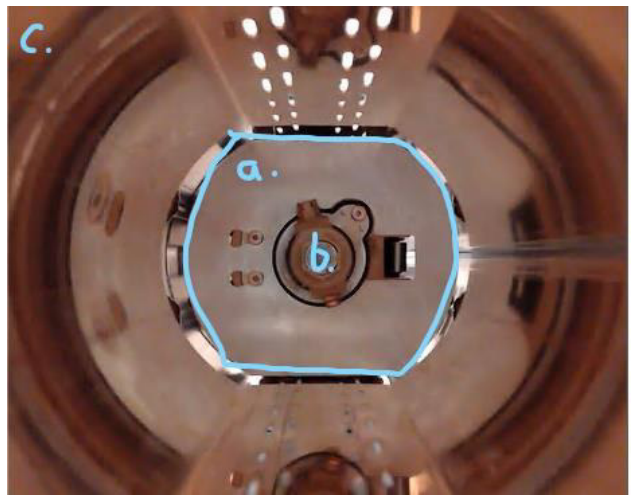

Figure 1: Longitudinal view of the inchworm inside a beam screen. The laser nozzle in the center points upwards. a. inchworm, b. nozzle, c. beam screen.

This setup requires a control system that communicates with each component and allows flexible parameter changes. The system must be reliable enough to run for many days unattended. For example, in a spiral treatment format (described later) we would need to treat 16m of beam screen while advancing by 50µm approximately every 5s. This would take up to 3 weeks. Similar times are expected for other sequences.

The system must also manage concurrent communication with all the components which make up the system and ensure that any issue is either resolved, or the system is safely stopped so the treatment can continue once the issue is resolved.

Movement of the inchworm makes up the bulk of the expected issues because of its mechanical nature. The system must be able to manage and identify movement problems and fix them where possible without affecting the overall process.

HARDWARE

The system consists of multiple hardware components connected to an NI CompactRIO (cRIO) real-time embedded system for control and monitoring. A cRIO was chosen because of successful implementations of cRIO-based control systems for other projects and because it provides a relatively straightforward programming model through LabVIEW.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV039

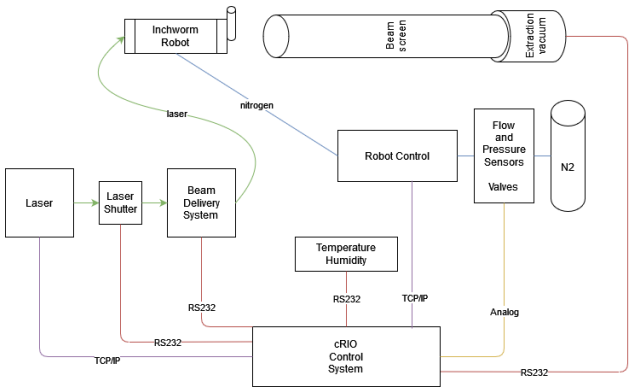

Figure 2: Main hardware topology.

Chief among the devices to control are the laser system and the mechanical robot that moves in the beam screen to distribute the laser to the surface. All hardware is connected to the cRIO through either TCP/IP connections, RS-232 or analog signals (see table 1 and figure 2).

Table 1: Connected Devices

Device	Connection
Robot	TCP/IP
Laser	TCP/IP (Telnet)
Laser shutter	RS-232
Laser BDS	RS-232
Extraction vacuum	RS-232
Temperature / humidity	RS-232
Flow and Pressure	Analog NI 9205
End switches	Analog NI 9205
Nitrogen supply valves	Relay NI 9481

The inchworm robot's movement scheme deserves some further description because it is at the core of how movements are defined in the software.

The robot is designed to move somewhat like a caterpillar and is composed of a fixed frame which contains a mobile internal sled. Both the frame and the sled can be clamped by pushing against the beam screen, which fixes its position. By activating the right clamps and moving the sled in the right direction, we can create a sequence of moves which advances the whole robot longitudinally along the beam screen.

The laser and its distribution, beyond an initial configuration of its settings, mostly turns into an on/off system. Sequences will use this facility to activate and deactivate the laser so that the treatment happens at the right times.

A nitrogen supply is used to pump nitrogen into the beam screen at the treatment point. Previous research has shown nitrogen to be an effective atmosphere for treatment of the surface [3]. Because the nitrogen tanks will run out before the full treatment is done, there are in fact 2 tanks. When the pressure from one tank falls below a defined threshold, the system automatically switches to the other tank and emails the operators so that the inactive tank can be switched out.

ARCHITECTURE

Control System Requirements

Control systems typically acquire many data points from various connected sensors and devices. Software control loops then use the acquired data to make decisions, which will affect the software itself as well as control actuators.

We decided early on that an architecture in which data would be at the center was the way to go. This leads to data being shared mostly globally in the application, and while shared data is often regarded as a danger, the small scope of the software implementation led us to accept this as a useful concept so that software procedures could more easily integrate a cross-section of application and hardware functionality.

Application Architecture

The go-to template for most LabVIEW applications over the past decade is the queued message handler (QMH). While QMHs promote modularity, and, when done properly, encourage code reuse, they can also make some applications needlessly complicated. Debugging and reading such code on embedded systems suffers from the many layers between an action from a GUI and the code that ends up running as a result. This is especially the case when frameworks force or encourage the use of re-entrant VIs, which cannot be debugged using LabVIEW's traditional debug tools, these tools being one of the main benefits of using LabVIEW. Because of this limitation and the small size of the application, it was decided to restrict the use of frameworks and to focus on a more direct and 'simple' approach to programming, in which events and their reactions are closer together.

Table 2 summarizes some of the most widely used LabVIEW frameworks, evaluating them in terms of readability, debuggability, prototyping ease and whether they can be easily instantiated multiple times.

Table 2: Framework Comparison

	Readability	Debuggability	Prototyping	Multiple instances
CVT [4]	yes	yes	yes	No
DCAF [5]	With experience	Not directly	Takes planning	Yes
QMH [6]	Can be	With experience	Nothing built-in	Yes
DQMH [7]	Lots of boilerplate code	Good testing tools	Scripting tools for quick creation of functions	Cloneable modules share some resources
Actor Framework [8]	With experience	Difficult in LabVIEW real-time	Slow to deploy	Yes

The application architecture includes a set of a few main processes (external communication, event handling, writing to file) and a series of monitoring loops which read from all the devices connected to the system. The monitoring loops mostly only read data into the system and make it available to the rest of the processes.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV039

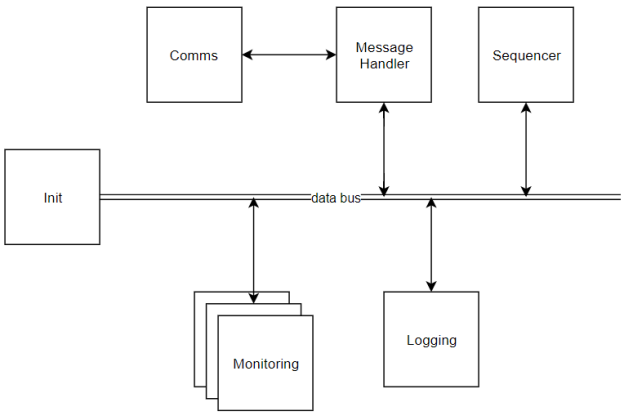

Figure 3: Software architecture.

There are few processes in the application and their roles are described here (see figure 3):

Message handler: Handles all incoming messages from the GUI. Any time the operator performs an action on the GUI, it gets sent to this process and for the most part is handled directly.

Logging: As its name implies, logs run-time data to file every few seconds so longer processes can be followed-up on in case of issues.

Monitoring: Many monitoring loops are implemented, one per device connected to the system. These processes gather all the needed data points from all devices and stores them in the central data storage for use by other processes.

Sequencer: The sequencer processes only run on-demand, and only one at a time. They simulate a series of requests and commands from the GUI so that the system can perform treatments over several days or weeks. Sequences are described in more detail further on in this paper.

IMPLEMENTATION

Because we are not using an established framework, it is important that the required attributes are built in from the beginning. The main components and attributes will be covered in this section.

Security and Safety

Like all control systems, we need to make sure that the system only does what it is supposed to. Any behavior that is outside of that established explicitly should be caught and the system should be put into a safe state. Indeed, if any behavior has not been explicitly planned for, we assume it is wrong and stop operation.

Because the movement of the robot is purely horizontal, we do not need any special consideration when stopping movement as it will simply stay in its position if we cut power to the drives.

The laser, being the active agent of this system needs to be considered more carefully because it will be damaging if it does not switch off when needed. There are multiple ways to prevent laser damage in this system. The laser can be powered off by a software command or its integrated shutter can be closed. Since a communication issue with the laser would prevent either of these safety mechanisms to be used, there is also an external laser shutter which can be independently controlled.

The system implements a safe state, which is a series of commands that is run when something unexpected happens. The main job of this routine is to stop all movement and to close all laser shutters. It puts the system into a safe mode from which it will only recover when the operator decides it is safe to do so.

Error Handling

Error handling is especially important in long-running embedded systems since operators are not monitoring the system 24/7. All errors need to be caught and, unless a recovery procedure is known, they must immediately put the system in a safe state.

Since there are many external hardware elements to be monitored during the whole treatment procedure, we make sure that communication to each device stays open and available at all times. Any communication issue immediately puts the system in its safe state.

Data Transfer and Communication

Because we wanted to keep a more direct path between events and their reactions, all processes in the application can use the central data store. This enables all processes to act in whichever way they deem necessary but introduces a higher risk of running into race conditions.

To avoid race conditions, we have a good definition of which process writes to which data point. Obviously, for data coming from external devices, only the respective monitoring loop writes that data.

All data in the application can be separated into 3 categories depending on how that data should be handled when restarting the application.

Configuration data is read-only data that comes from a configuration file on disk. This represents fixed configurations that do not change and which the application assumes will never change. We find such information as the radius of a beam-screen or the hardware address of an external device to control.

Settings represent data that the user can modify, and which can be saved to file. This allows the operator to define the details of the treatment and how the attached devices should run. These values will be remembered at subsequent launches of the application.

Run-time data is the collection of all other data the application keeps track of while it is running. Most of the data is contained here. The publishing process saves the relevant data points to a file during operation.

All 3 of these data sets are passed around the application to all processes and constitute what the application calls the data environment.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV039

Environment

The environment contains all the data needed for the various processes of the application to run properly. It is passed around as a Data Value Reference (DVR) which allows concurrent access throughout the application. One of the reasons for choosing to store this data like this rather than using existing solutions like the CVT (Current Value Table) is that this solution allows us to potentially run multiple instances of the application on the same hardware. We currently only run a single instance, but there were discussions at the beginning to run multiple systems from the same cRIO.

Giving each process access to the full dataset of the application means that each process can be more intelligent in making decisions because it has more context.

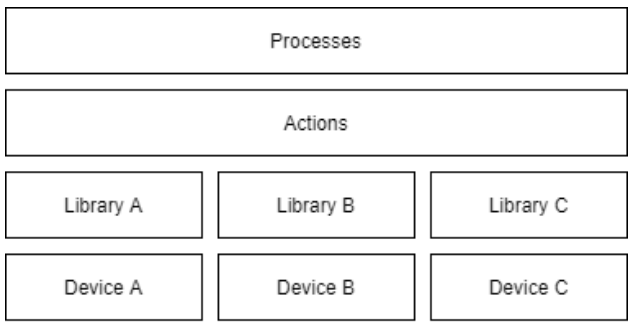

Figure 4: Application layers.

Actions

Each external hardware component is represented in the software by a library which encapsulates the communication with the device. These functions have no knowledge of the application they run in and therefore cannot be relied on to coordinate anything but the communication with that device.

To integrate these devices into a complete system, an extra layer is used above these libraries to coordinate them with each other. The first such layer is called the action layer. Each action takes in the environment data as an input and does something that the system needs to do (Figure 4). This can range from a low-level encapsulation of a single command to a single device, to a more complex one which reads data from multiple devices and coordinates their actions. The processes box represents the processes described at the end of the architecture description (monitoring, logging, communicating processes…). The processes mainly use the aforementioned actions to perform their tasks.

Sequences

To run the system autonomously, it is necessary to have some controlling process that sequences the steps necessary for a treatment run. Such processes are called sequences and they are only run on-demand when the operator sends the command. In figure 4 they run at the top-level processes category and have full access to the environment.

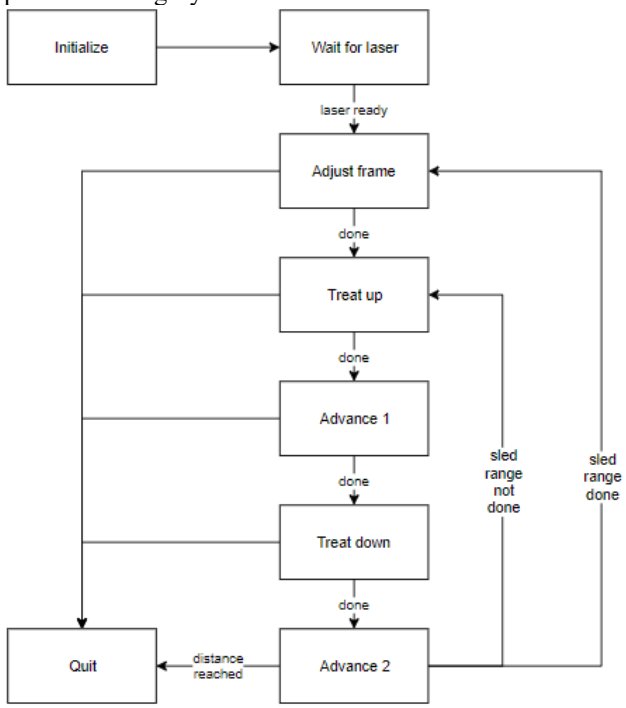

Figure 5: Line sequence logic.

More than one sequence is designed, but only one can ever run at any given time. The sequence takes control of the system and uses the data and actions to run it through the steps as the sequence defines them. When a sequence is started, the communication interface ignores most commands from the GUI because they would otherwise interfere with the sequence. In fact, sequences can be thought of as equivalent to the GUI + operator. They use the same actions as the GUI commands and simply wrap them up into a process which waits for events and reacts to them.

Sequences must also be implemented such that issues with the movement of the inchworm robot are properly handled. At each movement stage of the robot, care is taken to test for various outcomes, like getting stuck, so that it does not continue treatment.

Multiple sequences have been implemented to test their relative treatment effectiveness and are summarized here:

Spiral, as its name implies, moves the laser nozzle in a spiral fashion, in which it slowly advances while rotating the nozzle. Settings such as speed or spiral pitch allow the operator to make it more or less loose.

Line creates a movement similar to the spiral sequence, but where the forward motion is decoupled from the rotation. In this configuration it rotates with treatment, then advances without treatment, then rotates again to continue the treatment (Figure 5).

Longitudinal motion is a variation of the line sequence in which the treatment is done while moving linearly. Once it reaches the length to treat, it rotates slightly and starts treatment again.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV039

Zigzag is a spiral sequence variation in which it does not spiral out in a single rotational direction. The rotation goes back and forth while advancing the linear stage.

Move does a simple linear move of the robot to a new position and does not involve any treatment. It is useful for movements longer than the range of the robot sled because it sequences the necessary clamping and move details.

User Interface

The user interface runs on a separate PC connected to the same network as the other devices of the system. It connects to the 'event handler' process on the cRIO to send commands to the system. The event handler simply executes the commands as they are received if no sequence is running.

Communication between the PC and the cRIO is made with the AMC (Asynchronous Messaging Communication) library [9], which allows multiple systems to send messages to each other using a UDP connection.

Commands are sent from the PC to the cRIO to execute the above-mentioned actions and sequences. All data from the environment is constantly sent from the cRIO to the PC once it is connected so that it can display the current state of the system and help the operator understand what is currently happening.

Reliability

Because the system needs to run for days at a time with minimal intervention, it is extremely important that it performs reliably. Several features of the software implementation facilitate this and were mentioned throughout the paper. The main points are summarized here for convenience.

The GUI can be connected and disconnected from the device without affecting it, thus keeping it out of the loop for the core functionality.

A safe state has been implemented which sets all hardware to a desired state. Any unplanned error puts the system in this state so that an operator or expert can check the system before running again.

Because there are so few layers of software, actions are closely linked to triggering events, leading to fewer chances for errors and mistakes to appear.

Preliminary reliability tests in the absence of full hardware integration have shown that it is able to run a sequence reliably for 2 days, after which it was stopped because it performed dozens of cycles of the sequence without issue, as indicated by the event log file. As long as the laser isn't present, a longer test won't yield more useful results at this time. Further tests are planned when the full system will be assembled.

CONCLUSION

We described the architecture and implementation choices used for the project. A custom-made data and communication structure helped simplify the overall complexity of the software so that we could focus more on direct readability and benefit from LabVIEW's built-in debugging tools.

While the system is not yet operational (not all components are ready for integration), we have benefitted from the structure of the code, allowing for quick debugging while testing integration and sequences. Because of the smaller code footprint, it is also much faster to deploy and test variations of the software while fine-tuning behavior.

It is also important to note that while this structure is used successfully in this project, we must stress that it is very likely to show its limits if functionality were to get bigger and more complex. Its application in this project was made possible by the fact that no added complexity is foreseen. Another reason why we could safely rely on a simpler architecture is that if the needs do arise, we can quite easily replace this simple structure with something more complex.

FUTURE IMPROVEMENTS

As it stands, the project is ready for integration of final components, and it is not expected that anything big will need to change. Future improvements to the system will likely be in the details and timing of the sequences, which we've modularized to a simple state machine architecture.

REFERENCES

[1] M. Sitko *et al.*, "Towards the implementation of laser engineered surface structures for electron cloud mitigation", in *Proc. IPAC 18,* Vancouver (BC), Canada, Apr.-May 2018, pp.1220-1223. doi:10.18429/JACoW-IPAC2018-TUZGBE3

[2] R. Valizadeh *et al.,* "Reduction of secondary electron yield for E-cloud mitigation by laser ablation surface engineering", Appl. Phys. Lett. 105, 231605 (2014).

[3] S. Calatroni *et al.,* " Optimization of the secondary electron yield of laser-structured copper surfaces at room and cryogenic temperature", Phys. Rev. Accel. Beams 23, 033101 (2020).

[4] https://forums.ni.com/t5/Reference-Design-Content/LabVIEW-Current-Value-Table-CVT-Library/tap/3514251

[5] https://www.ni.com/en-us/innovations/white-papers/18/introduction-to-the-distributed-control-and-automation-framework.html

[6] https://www.ni.com/en-us/support/documentation/supplemental/21/using-a-queued-message-handler-in-labview.html

[7] https://delacor.com/products/dqmh/

[8] ni.com/actorframework

[9] https://forums.ni.com/t5/Reference-Design-Content/Asynchronous-Message-Communication-AMC-Library/ta-p/3494283