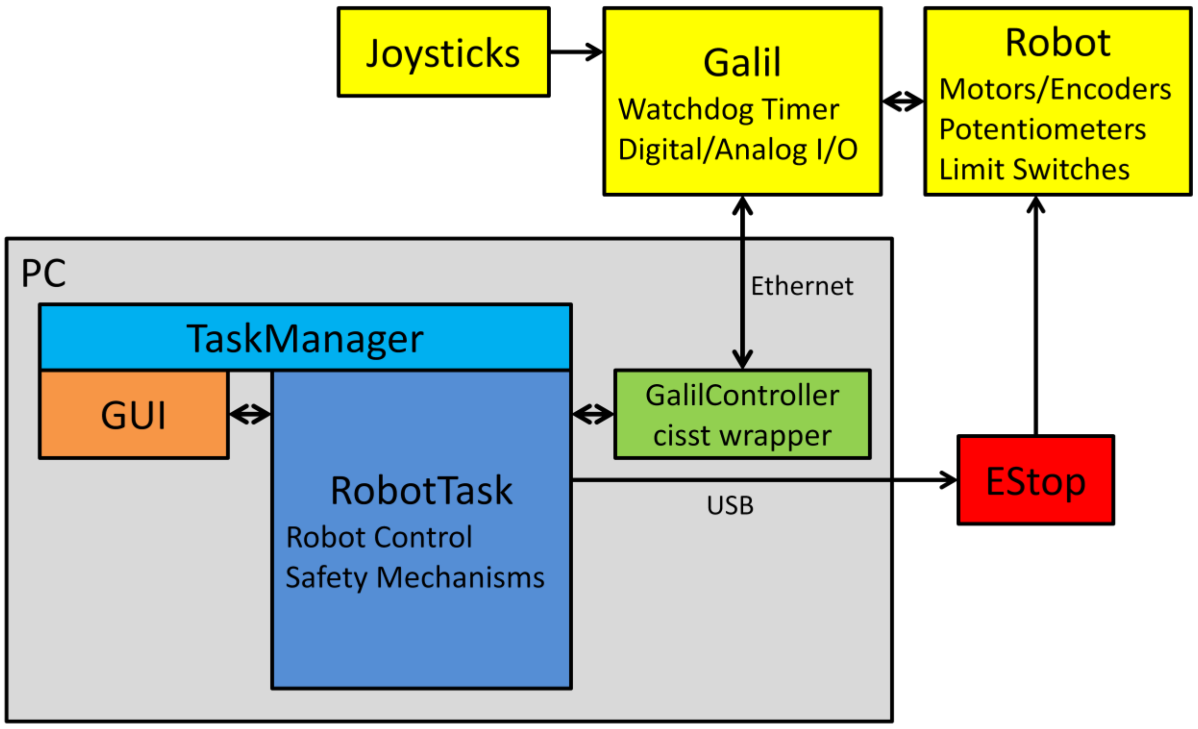
RoboELF Software Description

Software Overview and Dependencies

The software system for the RoboELF provides functionality to control the RoboELF and implement safety mechanisms. It is built around several existing software packages. The main applications make extensive use of the CISST libraries[1][2]. The CISST libraries are a set of software packages developed at the CIIST ERC at Johns Hopkins University. They include software designed for medical robotics and other computer integrated surgery applications. The GUI for the system is a Qt application[3]. Qt is a software package and development suite available from Digia. It provides a platform for creating graphical user interfaces. The RoboELF software system also makes use of the Galil C API and drivers to communicate with the Galil motor controller, a commercially available, 3-axis motor controller purchased from Galil Motion Control[5]. They provide a C-based software interface and hardware drivers to interact with their motion control products. The RoboELF software is designed to be built using CMake[6], a cross-platform open-source software building system. The RoboELF software currently runs on a laptop running the Ubuntu operating system[4], but can easily be extended to operate on other linux-based systems or on Windows. It has no other software dependencies.

The software interacts with the hardware of the system in several places. The PC interacts with the Galil motor controller to gather sensor data and issue motion commands. The Galil receives information from encoders, potentiometers, input switches and other electronic sensors. All of the data is accessible to the PC program. The PC also interacts with the USB relay that controls the Emergency Stop switch. To perform control of the robot, the PC program sends commands to the Galil, which implements them using a PID control process.



Software Workflow

The RoboELF software is built using the CISST library's component-based design pattern and Multitask libraries. The software system contains three major component classes, `devGalilController`, `robotTask`, and `qtDisplayTask`. `devGalilController` is the CISST C++ wrapper for the Galil motor controller and inherits from `cmnGenericObject`. It contains functions to send commands to, and receive feedback from, the motor controller by calling to the Galil C++ API. It is the low-level software connection between the PC and the motor controller. `robotTask`, which inherits from `mtsTaskPeriodic`, contains an instance of the `devGalilController` class along with all of the functions to implement safety and control algorithms for the system. `qtDisplayTask`, which inherits from `QObject` and `mtsDevice`, is the GUI portion of the system. It contains a Qt class, `throatGUI`, that implements the display structure. It also contains CISST multitask functions to receive data from `robotTask` and display it in realtime on the GUI. There is also a small `main` program that instantiates and initializes all of the other components.

The general lifecycle of the system is as follows:

1. `robotTask` and `qtDisplayTask` objects instantiated and connected via CISST multitask interface.
2. `Startup` and `Configure` functions called on both task objects to execute setup and calibration. This is where `devGalilController` and `throatGUI` are instantiated.
3. The `Run` function in `robotTask` is called at a set interval(50ms) until the program is closed. This performs all of the control and safety procedures to run the system.
4. When the GUI window is closed, the `Cleanup` and `Kill` functions are called on all tasks. This shuts down and ends all task processes.

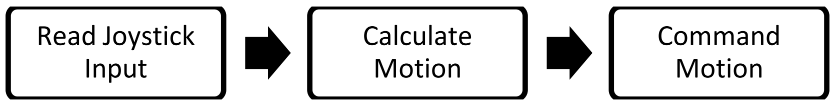
Hardware Interfaces

The Galil motor controller communicates to the PC program through a standard Ethernet TCP/IP connection. Information is transferred in 512-byte packets. A software watchdog timer is implemented to prevent uncontrolled motion in the event of a lost connection.

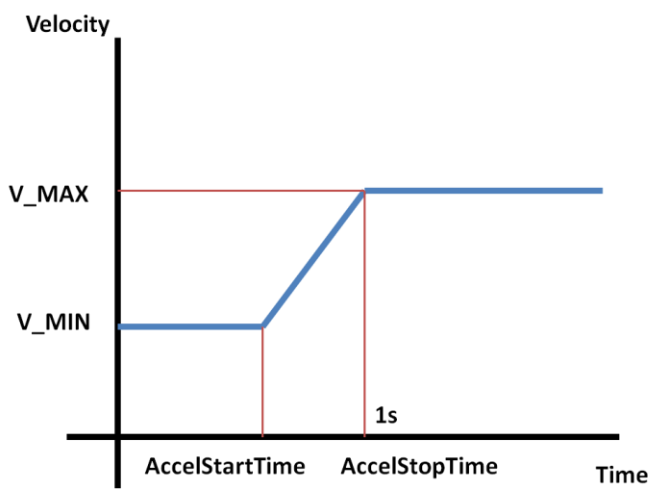
The PC connects to an Emergency Stop switch through a USB-powered relay. The relay behaves like a standard serial port device. The connection to the relay is checked every run loop to verify that a good connection is still active.

Robot Control Algorithm

The robot control function contains three basic steps:



These steps are implemented in the ManualControl function. The theory behind the control function is to perform incremental position control by continually commanding relatively small position move commands with the assumption that a new command will be issued before the goal position of the current command is reached. A desired velocity is also specified with each position command at which the robot should move to the commanded position. For the most part, the robot moves at constant velocity. However to give a smoother start when motion is first commanded, at the start of motion, the velocity is ramped up over a specified acceleration time window. This window is defined by the start and end time-boundaries where the time is measured from the time at which the motion command was first detected on the joystick input. At the beginning of motion until the start of the acceleration window, the robot moves at a minimum velocity. Between the boundaries of the acceleration window, the robot moves with constant acceleration until it reaches its maximum velocity. It continues at this maximum velocity until a neutral input is detected. If at any time the input changes to neutral, motion stops.

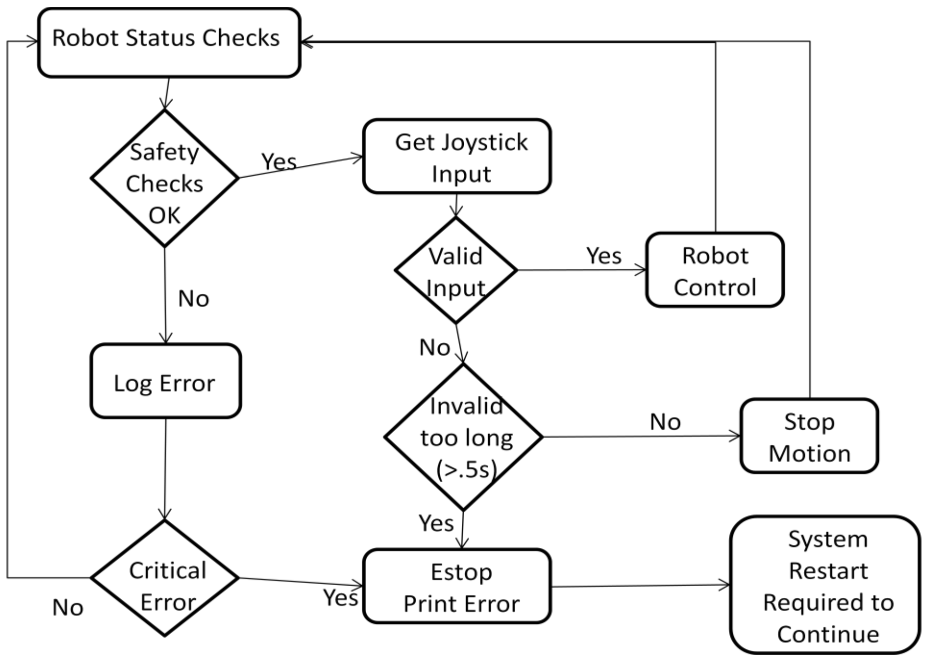


Safety Mechanisms

Most of the safety mechanisms are present in the software. They include: a software-activated emergency stop switch, a watchdog timer between the PC and the motor controller, redundant checking between motor encoders and potentiometers, and hardware and software motion limits. The emergency stop immediately cuts power to the motors and can be triggered by the software system via a USB activated relay switch or by pressing a large red button. This is the mechanism used to stop the robot if any of the other safety checks fail. The watchdog timer runs on both the motor controller and the PC, checking that each one can communicate with the other. There are digital encoders on the motors and analog potentiometers attached to each axis of motion. The values read from each of them are continuously compared to verify that the encoders are functioning properly. If any of the software safety systems show an error, or the Galil controller shows an error or failure, the emergency stop is activated and an error message is displayed. This message contains information about the error and the proper course of action to take to recover from it.

All safety failure detections are reported through exceptions, implemented using CISST's `cmnThrow` function. All failures are recorded, often with more detailed error information, to a log file. All exceptions inherit from the exception class `RobotException` which is defined in `devGalilController.` Within the `devGalilController` class, several other classes of exceptions are defined to differentiate between types of errors. More exception classes are defined in `robotTask.` These classes are associated with errors and/or failures detected by the functions in `robotTask.` All exceptions are caught by reference(to retain inheritance structure) and rethrown until they are caught in one of the main CISST task functions(Startup, Configure, Run, Shutdown). There they are caught and handled according to their subclass.

Below is a diagram of the basic system safety check workflow:



Emergency Stop

The emergency stop is the key to successfully implementing the software safety checks. It allows the PC to cut power to the robot motors at any time. It is important to verify that the connection to the emergency stop itself is working properly at all times. The USB relays used to activate the switch are accessed like a serial device, which can be accessed in the same manner as accessing a file in C++. One-byte messages are sent to the relays to turn them on or off. These messages are sent in the functions: `EStop_Startup`, `EStop_EmergencyStop`, and `EStop_TurnOff` in `robotTask`. To verify that the USB connection is still active and accessible, the PC program attempts to reopen it during each execution loop. This check is performed in the `EStop_CheckConnection` function in `robotTask`.

Watchdog Timer

The watchdog timer verifies that the connection between the PC and Galil is still active. It is implemented through message-passing between the PC and the Galil. The time that each message is sent is recorded and if too much time passes before a return message is received, an error is thrown. Two time limits are implemented for two levels of errors: shorter limit that, when tripped, silently logs that the error occurred, but does not stop the system, and a longer limit that immediately triggers the emergency stop. The shorter time limit is 50% longer than the task period(75ms), and the longer limit is 2.5 times the task period(125ms). These limits were selected through testing to ensure both safety and system stability. The logic to implement the PC-side of the watchdog timer is contained in the `GalilWatchdog` function in `robotTask`. The logic to implement the Galil-side of the timer is contained in the "Watchdog.dmc" file.

Encoder / Potentiometer Checking

To perform the encoder and potentiometer checking, a table of potentiometer values is read and stored during system startup. They can either be generated by running the calibration procedure or read from a file of old calibration results. The calibration procedure involves moving each axis of the robot through its full range of motion and recording the potentiometer values at a series of points along the axis. The points are identified by the encoder value at that position. Each point is an equal number of encoder counts away from the points around it, and the encoder value at the starting point is recorded. These two pieces of information can be used to calculate the encoder value corresponding to any point in the table of potentiometer values, therefore it is not necessary to store a table of encoder values. There are a number of specific checks during calibration and during runtime to verify the potentiometer values. When values are read during calibration, the potentiometers are checked to have monotonically increasing values over their range, and the total range of the potentiometer input is checked against known values to ensure they are accurate. During runtime the current encoder value is used to determine an expected potentiometer value by finding the value in the table that corresponds to the current encoder value. Linear interpolation is used to calculate expected values that fall between table entries. The expected potentiometer value is compared to the current potentiometer value, and if the difference is greater than a certain threshold a safety error is thrown.

Galil Over-Voltage Check

The Galil has built-in checks that cut power to the motors if too much voltage is applied to an axis motor while no motion is detected on that axis. This will stop motion in the event that some object is blocking the axis from moving or if some other mechanical malfunction is preventing motion. This function does not send any error message to the PC when the error is detected. To check for this error, the PC program reads the status of the motors during each execution loop. If any of the motors turn off unexpectedly, it is recorded as a safety failure. This check is performed in the `Check_MotorsActive` function in `robotTask`.

Input Joystick Checking

The input joysticks each control two switches, the input of which are recorded by the Galil and queried by the PC program. In the PC program the inputs are read and interpreted as motion commands using a lookup table of switch states. The two switches provide redundancy that allows greater detection of errors. Any single-switch failure is detectable with this design. It is a safety failure if the joystick switches are in an invalid state. This design also allows false positives to occur in the error checks. Both switches do not throw simultaneously, so it is possible to record an input during a joystick transition that registers as an invalid state. Therefore, the check is implemented as an allowable length of time that the switches can be in an invalid state(the transition period). An error is triggered if the joystick switches are in an invalid state for longer than .5s. The joystick input reading and checking is performed in the `GetJoystickInput` function in `robotTask`.

Class-by-Class Functional Description

Note: Mask parameters. Many functions take “mask” parameters to indicate on which axes commands should be applied. Wherever they appear, they behave in the same way. The mask variable is a Boolean vector containing as many elements as there are motors axes on the robot. The command is applied to all axes with a corresponding mask value of “true”.

robotTask

robotTask is the class that contains all of the RoboELF-specific functions and data for the software system. It is a CISST mtsTaskPeriodic. It contains functions to implement safety mechanisms and perform high-level control operations.

Inner Classes(Exceptions):

`EStopException`

This is the exception that is thrown when the connection with the Emergency Stop fails. It is generated by any of the EStop functions.

`EncoderException`

This is the exception that is thrown when the encoder/potentiometer check fails during a run. It is also thrown if a failure is detecting during the calibration procedure.

`MotionException`

This is the exception that is thrown when it is detected that the Galil has tripped its overvoltage limit and unpowered a motor. It is thrown by the `Check_MotorsActive` function.

`WatchdogException`

This is the exception that is thrown when the watchdog timer runs out, indicating a lost connection to the Galil. It is thrown by the `GalilWatchdog` function.

`InvalidInputException`

This is the exception that is thrown when an invalid joystick position is held for a longer than allowable time(.5s). It is thrown by the `GetJoystickInput` function.

Important Class Members:

Enums:

`REV`, `STOP`, `FWD`

Indicate desired motion states of the motors, based on joystick state. The variable `CurMotion` is set to one of these values in `GetJoystickInput`.

`MLIMFWD`, `MLIMREV`

Identify the array index of forward and reverse software motion limit values.

`OPMODE`

Used as a flag to turn on or off or change the behavior of debugging and logging features for development or clinical use.

Static Members:
(* indicates the variable is settable via config file)

`NB_Actuators`

The number of joints on the robot.

`JOYSTICK_INPUTS`

An 6x2 array of integers that contains all valid pairs of joystick states. `GetJoystickInput` compares the current state to this table to check validity.

`NO_INPUT`

The integer code indicating joystick neutral position.

`ESTOP_ON/OFF`

The integer codes that should be sent to turn the Emergency Stop switch on or off.

`ALLOWED_ERROR`

The amount of error, as a decimal fraction, allowed in checking encoder/potentiometer values during runtime.

`INPUT_TIMEOUT`

The timeout value for invalid joystick input.

`MIN/MAX_VEL`

The minimum and maximum velocity of each axis of the robot.

`ConfigFileName`

The name of the configuration file containing values for many static constants. Default is "RoboELF.config".

`TableStepSize` *

The distance in encoder counts between each sample in the encoder/ potentiometer checking table.

`AnalogRange` *

The approximate range of the analog potentiometers for each axis.

`VoltageLimits` *

The limit voltage that the Galil's overvoltage protection should be set to.

`AccelStart/StopTime` *

The times defining the acceleration window in the velocity function.

`EStop_PORTNAMES` *

The names of the ports that the Emergency Stop can be plugged into. The value of `OPMODE` determines which name is used.

`GalilDefaultIP` *

The default IP address of the Galil. The program attempts to connect with this address before prompting to select from a list.

`GalilPIDGains` *

An array containing the PID gain values for the Galil.

Non-static Members:

`Timer`

CIIST Timer object used by all of the time-dependent functions including the watchdog and control functions.

`ActuatorState`

Variable containing the current state of the robot. Includes information such as encoder position, limit switch state, and motor state.

`AnalogInput`

Variable containing the current state of the analog potentiometers.

`Encoder/AnalogValues`

Variables containing the current state of the encoder and potentiometers.

`AnalogPrev`

Variable containing the previous recorded potentiometer value for comparison purposes during calibration.

`HomeOffset`

The offset in encoder counts required to treat the minimum range of the motors as the zero position.

`TableRange`

The total range in encoder counts of each motor axis.

`InputMismatch`

Flag indicating a joystick error has occurred.

`MotorLimits`

A 3x2 int array containing the forward and reverse motor limits in encoder counts.

`Galil/GalilConnected/GalilIP`

Galil is the instance of devGalilController used to interact with the Galil. GalilConnected is a flag indicating that a good connection is active. GalilIP is the IP address of the connected Galil.

LastGalilWatchdog

The time of the last received watchdog response from the Galil.

Watchdog(Non)CriticalTimeout_Galil

The timeout values in milliseconds for critical and non-critical watchdog timeouts.

WatchdogDisable_Galil

Flag indicating the watchdog timer has run out.

CurMotion

An int 3-vector containing the current state of motion of each motor axis, REV, STOP or FWD.

StartTime

The time that the current motion command was first detected on the joysticks.

ACCEL

The rate at which the robot will accelerate on each axis, calculated from the min/max velocity and acceleration window for each axis.

InputIsValid

Flag indicating that the joystick input is valid.

InvalidInputTimeoutPeriod/FirstInvalid

The length of the invalid joystick input timeout and the time at which the last invalid joystick position was detected.

EStop_PortName

The name of the Emergency Stop port to be used.

EStop_IsConnected

Flag indicating that the Emergency Stop is connected.

`SafetyOneShot`

Flag indicating that a safety failure has occurred. If this flag is set to true, the program will run until the user manually closes it, but will not run any control or safety functions until restarted.

Important Class Functions:

`robotTask`()

Initializes non-static variables. Creates an instance of `mtsInterfaceProvided`, "ProvidesThroatRobot" which is used to provide information to `qtRobotDevice`. Reads configuration file by calling `ReadConfigFile`, which reads the file and sets the appropriate variables.

`Configure`()

Contains most of the startup and calibration routines. It opens a connection with the Emergency Stop, turning it on by calling `EStop_StartUp`. It starts a connection with the Galil by calling `Galil.Init`(). It sets several variables on the Galil including the command timeout period, the amount of time to wait for a response to a sent command, to 20ms. It configures the extended IO to the correct setting by sending the command "CO 0". It sets the PID gain values, sets and activates the Galil's overvoltage and position error safety limits. The values for all of those settings are read from the configuration file. It also sets the Galil's motor acceleration values to ten times the maximum velocity. This allows the assumption to be made that acceleration of the motors up to their commanded speed is instantaneous. Finally, Configure prompts the user to select a calibration method and calls the appropriate function to run calibration. If the user input is invalid, it requests new input until valid input is received. All of the functionality in Configure is contained in a try-catch block to handle any safety, or other, exceptions. Exceptions are caught and handled as described in the Safety Mechanisms section.

`Startup`()

Starts the watchdog timer between the Galil and PC. It downloads the watchdog program to the Galil by calling `Galil.ReadConfigFile`() and runs it by sending the command "XQ #wchdg". If all startup procedures have completed successfully, verified by checking the `SafetyOneShot` flag, it prints "RoboELF Ready for Use" to the command line output.

`Run`()

This function contains all of the normal runtime operations including safety checks and robot control. It is the CISST runtime function that is called repeatedly at a specified interval(50ms). It first runs all of the safety checks by calling `RunSafetyChecks` then if all checks passed, it calls `ManualControl` to run the control algorithm. If any safety failures occur during operation, the exception is caught in Run's catch block. All failures are logged and serious failures trigger the Emergency Stop.

`Cleanup`()

This is the CISST function that is called when the system is shutdown. It turns off the Emergency Stop and closes the Galil connection. If an error occurs, it is logged and any shutdown functions are continued if possible.

`Home`()

This function performs calibration procedures to build the tables that are used for encoder potentiometer checking. It first configures several settings on the Galil in preparation for calibration. It sets the desired velocity for all axes to their maximum and sets the software limits for each axis to the maximum possible value, effectively turning them off for calibration. It then moves the axes to their default zero position. Depending on the user input selection for calibration method, it rebuilds the calibration tables by calling BuildTables or reads them from a file by calling BuildTablesFromFile. If it does not find a valid calibration file, it rebuilds the tables normally. After completing calibration, it sets the software limits on the Galil to the values found during the calibration routine, either read from file or found during normal calibration. If calibration completes successfully, it prints "Calibration Successful". If calibration failed for any reason, it rethrows the exception.

`BuildTables` ()

This function implements the calibration procedure for the robot. The purpose is to build tables of potentiometer values that will be used to perform the encoder/potentiometer safety check during operation. A table is built for each motor axis. The values are stored in AnalogValues. A secondary purpose is to find the software motion limits for each axis, which are set after calibration is complete. The same procedure is followed for each axis, one at a time:

The axis is moved to the negative hard motion limit, then back to just off the limit. This position is saved as the reverse software limit and the first position in the table. Next the axis is moved forward incrementally until the forward hard limit is hit. After each incremental move the position is recorded in the table. The size each move is determined by TableStepSize, which contains a step size value for each axis. When the forward motion limit is hit, the axis moved back to just off the limit. This final position is saved as the forward software motion limit.

Several checks are implemented to ensure safety during calibration and verify that the calibration is accurate. The Galil overvoltage and position error checks are active during calibration and if either one is tripped, calibration is aborted. To verify that the potentiometers are giving accurate results, each new value is compared to the previous recorded value to check that values are always increasing over the length of the axis. At the end of calibration, the total range of the table is compared to a saved constant value. If the results of calibration are not consistent with the expected value, an exception is thrown and the run does not proceed. After all axes have been calibrated they are moved to the center of the range of motion. The results of the calibration are written to a file so that they can be read in later to allow a quick calibration that does not require moving each axis through its full range of motion.

`BuildTablesFromFile`()

This function performs a faster calibration be reading previously obtained values from a file instead of building new tables. It assumes that the values in the file are a valid calibration and so does not check them but simply reads and saves them. It also reads the forward and reverse software motion limits from the file. It places all axes at the center of their range of motion just as the normal calibration does.

`ReadConfigFile`()

This function reads and parses the configuration file to set constant values at the start of operation. If the file is not found, an error is logged and the program continues, using default hardcoded values for the constants.

`ManualControl`()

This function implements the control algorithm of the robot. It reads the current joystick input, calculates and sets the motor velocity, and commands a move. The joystick input is read by calling `GetJoystickInput` which sets the values in `CurMotion`. These values determine in which direction each axis moves, if at all. The velocity calculation procedure is repeated for each axis:

If `CurMotion` is set to `STOP`, the desired velocity is set to zero and no further computation is necessary. If `CurMotion` is set to `FWD` or `REV`, the velocity is calculated based on how long the axis has been in motion according to the procedure described in the Robot Control Algorithm section. Once a desired velocity has been calculated, it is sent to the Galil. Then a position move command is made. The logic for sending the command is to set an absolute position goal that is the current position of the axis plus a positive or negative offset. A command is given even if no motion is desired. If no motion is desired, no offset is added to the current position. The offset is arbitrarily set to the calibration table step size. If a safety failure occurs at any point, the exception is rethrown and no motion is commanded.

`RunSafetyChecks`()

This function runs safety checks during operation including the encoder and potentiometer checking in `Compare_Inputs`, the watchdog time in `GalilWatchdog`, the Emergency Stop connection in `EStop_CheckConnection`, and checks for the Galil overvoltage and position error limits by calling `Check_MotorsActive`. If any of the safety checks fail, they throw an exception and/or set a flag variable. These flags are checked after all functions have completed. If any flags have tripped, the `SafetyOneShot` flag is set to true. All exceptions are rethrown up to the `Run` function.

`UpdateActuatorState`()

This function calls `UpdateActuatorState` and `GetAnalogInputs` in `devGalilController` to update the state of the robot.

`Enable/DisableMotors`()
`ActuatorWaitMotion`()
`ActuatorPositionMove`()
`ActuatorVelocitySet`()

These functions call the functions in `devGalilController` with the same or similar names to perform the indicated behavior.

`Compare_Inputs`()

This function implements the encoder/ potentiometer checking during operation. The same procedure is applied to all axes of motion. The first step of this check is to find where in the table we expect to be, based on the current encoder readings. This can be calculated by finding the current position past zero, assuming zero is at the negative motion limit of the axis, or HomeOffset, and dividing by TableStepSize. This gives the index of the table value that is closest to the current potentiometer value that is also less than the current value. Linear interpolation is used in between table values to calculate an expected value for the potentiometer. This expected value is compared to the actual value. If the difference is larger than (1+ALLOWED_ERROR) times the difference between the two closest table values(the potentiometer value step size), an EncoderException is thrown.

`GetJoystickInput`()

This function reads the input from the Galil and interprets it into motion commands. It also checks it to ensure that it is valid. First, it reads the current state of the Galil’s external IO, where the joysticks are wired. The first check it performs is to check if all joystick axes are in the neutral state by comparing their value to the `NO_INPUT` value. If they are in the neutral position, `CurMotion` is set to `STOP` to put all axes in the stopped state. The input is recorded as valid. If the first check does not pass, the input state is compared to all valid states in `JOYSTICK_INPUTS`. If a match is found, the state for the appropriate axis in `CurMotion` is set to `FWD` or `REV`. If no appropriate match is found among the valid codes, the input is recorded as invalid.

When an invalid input is detected, `CurMotion` is set to all `STOP`, the time is recorded in `FirstInvalid` and the `InputIsValid` flag is set to `false`. If the `InputIsValid` flag is already `false`, an invalid input has been detected before. The current time is compared to the time stored in `FirstInvalid` and the difference compared to `InvalidInputTimeoutPeriod`. If the time since the first invalid input is greater than the timeout period, an `InvalidInputException` is thrown. If a valid input state is detected after an invalid one, but before the timeout period has fully elapsed, the `InputIsValid` flag is set back to `true`, and normal operation resumes.

`EStop_Startup/TurnOff` ()

These functions respectively turn the Emergency Stop on and off by opening a connection to the USB relay which is accessed like a normal file and sending the appropriate code identified by the constants `ESTOP_ON` and `ESTOP_OFF`. These functions are used during system startup and shutdown. If the function is unable to open a connection to the relay, it throws an EStopException.

`EStop_CheckConnection`()

This function verifies that a connection to the Emergency Stop is still active and it can be accessed normally. It does this by attempting to open a new connection, which looks like opening a file. If the connection is successful, it closes it and continues normally. If the connection is not successful, it throws an `EStopException`.

`EStop_EmergencyStop`()

This is the function used to activate the Emergency Stop and cut power to the robot when a safety failure occurs. It sends the command to open the relay switch, disconnecting the motor power circuit and logs that the Emergency Stop has been activated. If it is unable to connect to the Emergency Stop, it disables the motors through the `DisableMotors` function and throws an `EStopExcepion`.

`Check_MotorsActive`()

This function checks the motor state to check if any of the motors have been disabled, indicating that a position or overvoltage error has tripped on the Galil. It throws a `MotionException` if it detects a disabled motor.

`GalilWatchdog`()

This function implements the PC side of the watchdog timer between the Galil and the PC. The first thing it does is check how long it has been since the last message was received from the Galil. If that time is greater than WatchdogCriticalTimeout_Galil, the timer times out, sets the WatchdogDisableGalil flag to true, and throws a WatchdogException. If the time is greater than WatchdogNonCriticalTimeout_Galil, it logs the non-critical timeout and continues. If the time is less than WatchdogCriticalTimeout_Galil, it continues the watchdog procedure as normal. It first sends a message to the Galil then checks for a response. If it receives a response, it checks if the response is normal. If it is, it resets LastGalilWatchdog to the current time and returns. If it receives an error response, it throws a WatchdogException. If it does not receive a response, it returns normally but does not reset LastGalilWatchdog. If any errors occur in communication with the Galil, it rethrows the exception.

devGalilController

`devGalilController` is the CISST wrapper for the Galil C API to communicate with the Galil motor controller. The Galil Command and User manuals should be consulted for details about Galil commands and behavior. Notably, it also contains the base class for all safety failure exceptions in the RoboELF software system, `RobotException`.

Inner Classes(Exceptions):

`RobotException`

This is the base class for all error exceptions generated in the RoboELF software system. It is itself a subclass of `std::runtime_exception`. It is never explicitly instantiated anywhere but allows other exceptions to be caught in a more general way. It also contains a virtual function `raise` that allows it to be caught and rethrown by reference, preserving inheritance relationships.

`ExcpCommError`

This exception is thrown when a communication error occurs while attempting to send or receive messages and commands from the Galil.

`ExcpSystemError`

This is a generic exception that is thrown when a minor error occurs, such as a Galil command error. These errors are regarded as non-serious by `robotTask` when they are caught, therefore it is simpler to aggregate them into a single exception type. Separate errors are still logged when they occur for maintenance and debugging purposes.

`ExcpMotionError`

This exception is thrown when a non-communication-related problem is detected in a motion command function such as `WaitMotion`, `SetPostionMove` and `SetVelocityMove`.

`ExcpPowerError`

This exception is thrown when an error occurs while attempted to turn the motors on or off.

`ExcpWaitMotion`

This exception is thrown when an errors occurs in the `WaitMotion` function.

Important Class Fields:

`galil`

Instance of the Galil API interface class.

`AnalogInput`

Vector containing values of the potentiometers on each axis.

`GalilIP`

String holding the current IP address of the Galil.

`NB_Actuators`

Enum defining the number of axis present on the robot.

Important Class Functions:

`Init`()

This is the startup function. It takes an IP address as a parameter that will be used to connect to a Galil via TCP/IP connection. If a Galil is not found at that address, a menu prompt will be displayed with available addressable Galil controllers. If a Galil is found and a connection is made, the IP address is saved in GalilIP. If connection fails or another error occurs, an exception is thrown.

`Close`()

This is the shutdown function. It turns off the motors and deletes the galil object.

`Enable`/`DisableMotorPower`()

These functions turn the motors on and off using the “SH” and “MO” commands, respectively. They can be applied to all or some of the motors by passing a mask parameter. If no parameter is given, the command is applied to all axes. If an error occurs, an `ExcpPowerError` is thrown.

`StopMotion/All`()

These functions stop motion, but to not cut power from, all or some of the motors as indicated by the mask parameter(`StopMotionAll` calls `StopMotion` with all true mask parameter).

`GetGalilMessage`()

This function retrieves any messages in the queue that have been sent from the Galil to the PC. If no messages are in the queue, it returns an empty string.

`GetActuatorState`()

This function reads all necessary values from the Galil to populate a `prmActuatorState` variable. It also reads the current state of the analog inputs. The actuator state values include, for each axis: in motion, home switch and motor off flags, forward and reverse limit switch status and a timestamp.

`GetAnalogInputs`()

This function places the analog values read from the Galil in `GetActuatorState`() into a passed-by-reference parameter variable. They are not returned in `GetActuatorState` because they are not part of the `prmActuatorState` variable.

`ReadExIO`()

This function reads the Galil’s external IO status. The external IO is configured in 4 blocks that must be read individually. `ReadExIO` takes a vctInt4 variable as a parameter to store the results and return them. To read each IO block, the command “TI [2,3,4,5]” is sent to the Galil(the blocks are numbered 2 through 5). The response to each command(the status of that IO block) is stored in the parameter variable. If an error occurs, a `ExcpSystemError` is thrown.

`SetPositionMove`()

This function is used to command motion with the Galil. It puts the Galil into Position Tracking mode which allows an absolute goal position(specified in encoder counts) to be given and updated on the fly while motion is already under way. The goal position is given as a masked input parameter. If there is an error, an `ExcpMotionError` is thrown.

`SetVelocity`()

This function sets the velocity at which each axis will move when given a motion command. The desired velocity is given as a masked input parameter. If there is an error, an `ExcpSystemError` is thrown.

`SetAc/Decceleration`()

These functions set the acceleration/deceleration with which each axis will move when given a motion command. The desired value is given as a masked input parameter. If there is an error, an `ExcpSystemError` is thrown.

`SetFwd/RevSoftwareLimits`()

These functions set the forward and reverse software motion limits, given in encoder values, on the Galil. By default these values are set to maximum range and do not affect motion. After they have been set the Galil will prevent the motors from moving past the specified limits.

`Set/Activate/DeactivateOverVoltageLimit`()

These functions set a value for, activate and deactivate the Galil's overvoltage safety limit. The Galil's overvoltage limit turns off a motor if it detects that voltage over a specified limit is being applied, but the motor is not moving. By default this feature is turned off and the default voltage threshold is about 1.5V. Note that this feature does not send any notification when it disables a motor. If an error is detected, an `ExcpSystemError` is thrown.

`Set/Activate/DeactivatePositionErrorLimit`()

These functions set a value for, activate and deactivate the Galil's position error safety limit. The Galil's position error limit turns off a motor if the current position is farther from the commanded goal position than a specified threshold. By default this feature is turned off. Note that this feature does not send any notification when it disables a motor. If an error is detected, an `ExcpSystemError` is thrown.

`SetGains`()

This function sets the PID gain values on the Galil. It takes the values as a one-dimensional int array organized in the following manner:
[kdA, kpA, kiA, kdB, kpB, kiB, kdC, kpC, kiC]

`ReadConfigFile`()

This function downloads a program file to the Galil's ROM. The file name is passed as a string parameter. If an error is detected, an `ExcpSystemError` is thrown.

`WaitMotion`()

This function blocks until the axes indicated by a mask parameter have stopped moving, or until a timer runs out. It loops every .002 seconds and interrogates the Galil to see if the given axes are in motion. It also checks if the timer has run down. If the timer runs out first, an `ExcpWaitMotion` is thrown. If motion stops, it returns normally.

`SendCommand`()

This function is used by almost every other function listed to send commands to the Galil. It calls the Galil API function to send a command to the Galil and returns the Galil's response. All commands and responses are sent as strings. If the Galil returns an error, `SendCommand` throws an exception, a `ExcpCommError` if it is an invalid command or timeout error, an `ExcpSystemError` it is anything else other than a command error. Command errors are ignored to reduce log file clutter. They occur most often when a motion command cannot be completed for a non-dangerous reason, for example if it is at a range limit.

`CreateCommand`()

This is a helper function that assembles string commands for axes specified by a mask parameter. It takes as parameters the command to send, a mask variable indicating which axes it applies to, and a list of axis-specific parameters to pass along to the Galil. It creates string commands of the form "<command> <axis A param>, <axis B param>, <axis C param>". For example "FL 10000,20000,30000".

`CreateComandForAxis`()

This is a helper function that assembles string commands for axes specified by a mask parameter. In contrast to the CreateCommand function, it does not pass axis-specific parameters to the Galil. It creates string commands of the form "<command> <axis><axis>…". For example: "SH ABC".

qtRobotDevice

Contains an instance of the GUI class and fields and functions to populate and update the GUI during runtime. It uses the CISST library's command pattern to get data from `robotTask`.

Important Class Fields:

`ActuatorState`

Holds information about the current robot state including encoder counts and limit switch state.

`AnalogInput`

Contains the current values of the analog potentiometers.

`GetActuatorState`
`GetAnalogInput`
`GetEncoderRange`
`GetEncoderOffset`

These are all CISST Multitask `mtsFunctionRead` functions that have definitions in `robotTask`. They read the indicated variables in `robotTask`.

`SliderValues`

The current position values for the dials and slider on the GUI.

`RobotGUI`

An instance of `throatGUI` that defines the GUI.

Important Functions:

`qtRobotDevice`():

Constructor initializes an instance of `mtsInterfaceRequired`, named "RequiresThroatRobot". This is what allows `qtRobotDevice` to communicate with `robotTask`. The constructor also initializes and starts the update time for the GUI.

`Configure`():

`Configure`() sets the size and position of the GUI and displays it.

`QSlotTimerUpdate`():

This function is called every time the update timer runs down. It refreshes the values on the GUI. It reads the current values from `robotTask` by calling its CISST Multitask functions and sets the displayed values to these new ones.

`ConvertSliderValues`():

This function is called in `QSlotTimerUpdate`(). It uses the encoder position and offset values read from `robotTask` to set the position of the dials and slider on the GUI. To display properly, these values must be scaled to a 0-100 scale.

throatGui

Contains one function, a constructor, that defines and initializes the GUI components. The GUI uses `QLCDNumber`s for all of the active display fields and `QLabel`s for all of the static text labels. It uses `QDial`s for the rotational axis displays and a `QSlider` for the linear axis.

References

[1] CIIST Libraries. CIIST ERC. http://cisst.org

[2] Kazanzides P., DiMaio S., Deguet A., Vagvolgyi B., Balicki M., Schneider C., Kumar R., Jog A., Itkowitz B., Hasser C., Taylor R. The Surgical Assistant Workstation (SAW) in Minimally-Invasive Surgery and Microsurgery. 2010 Jun.

[3] Qt. http://qt-project.org, http://qt.digia.com

[4] Ubuntu Operating System. http://www.ubuntu.com

[5] Galil Motion Control. http://www.galilmc.com

[6] Cross Platform Make(CMake). http://www.cmake.org