```
/*
 * File:   Longitudinal.cc
 * Author: lpiccoli
 *
 * Created on November 18, 2010, 13:30
 */

#include "Longitudinal.h"
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/vector_proxy.hpp>
#include <boost/numeric/ublas/matrix_proxy.hpp>
#include <boost/numeric/ublas/io.hpp>
#include <cmath>
#include <vector>
#include "epicsTime.h"
#include "ExecConfiguration.h"
#include "MeasurementAverage.h"
#include "PatternManager.h"

USING_FF_NAMESPACE

using namespace boost::numeric::ublas;

/**
 * Class constructor. Only initializes attributes.
 *
 * @author L.Piccoli
 */
Longitudinal::Longitudinal(std::string algorithmName) :
Algorithm(algorithmName),
_dl2EnergyActuator(NULL),
_firstCalculate(true),
_blenBc1Average(BLEN_AVERAGE_ELEMENTS, "BC1 BLEN Average"),
_blenBc2Average(BLEN_AVERAGE_ELEMENTS, "BC2 BLEN Average"),
_dl1Path(NO_MEASUREMENT),
_dl2Path(NO_MEASUREMENT),
_dl2AutoLimitControl(true),
_dl2HihiChannel(NULL),
_dl2LoloChannel(NULL),
_previousMode(false),
_checkMeasurementsStats(50, "Long: checkMeasurements"),
_selectStatesStats(50, "Long: selectStates"),
_configureMatrixStats(50, "Long: confMatrix"),
_calculateEnergyCurrentStats(50, "Long: calculateEnergyCurrent"),
_updateStatesStats(50, "Long: updateStates"),
_updateActuatorsStats (50, "Long: updateActuators"),
_updateActuatorsDetailStats (50, "Long: updateActuators - detail") {
    for (int i = 0; i < MAX_LONGITUDINAL_MEASUREMENTS; ++i) {
        _measurementStatus[i] = INVALID;
    }

    // Initialize current state of the FBCK PV to false (OFF).
    // The FBCK PV is initialized AFTER the configure() method
    // is invoked.
    for (int i = 0; i < MAX_LONGITUDINAL_MEASUREMENTS; ++i) {
        _actuatorFbckState[i] = false;
    }

    _energyAverage[DL1_ENERGY] = new MeasurementAverage(INTEGRAL_AVERAGE_COUNT,
"DL1 Energy");
    _energyAverage[BC1_ENERGY] = new MeasurementAverage(INTEGRAL_AVERAGE_COUNT,
"BC1 Energy");
    _energyAverage[BC2_ENERGY] = new MeasurementAverage(INTEGRAL_AVERAGE_COUNT,
"BC2 Energy");
    _energyAverage[DL2_ENERGY] = new MeasurementAverage(INTEGRAL_AVERAGE_COUNT,
"DL2 Energy");

    _currentAverage[BC1_CURRENT] = new MeasurementAverage(INTEGRAL_AVERAGE_COUNT
, "BC1 Current");
    _currentAverage[BC2_CURRENT] = new MeasurementAverage(INTEGRAL_AVERAGE_COUNT
, "BC2 Current");
}
```



```
Longitudinal::~Longitudinal() {
    for (int i = 0; i < INTEGRAL_ENERGY_ELEMENTS; ++i) {
        delete _energyAverage[i];
    }

    for (int i = 0; i < INTEGRAL_CURRENT_ELEMENTS; ++i) {
        delete _currentAverage[i];
    }
}

using namespace boost::numeric::ublas;

/**
 * Initialize algorithm parameters based on data stored in the LoopConfiguration
.
 *
 * @param configuration pointer to the LoopConfiguration
 * @return 0 on success, -1 on failure
 * @author L.Piccoli
 */
int Longitudinal::configure(LoopConfiguration *configuration,
        MeasurementSet *measurements, ActuatorSet *actuators,
        StateSet *states) throw (Exception) {
    _loopConfiguration = configuration;
    _measurements = measurements;
    _actuators = actuators;
    _states = states;

    Log::getInstance() << Log::flagConfig << Log::dpInfo
                       << "INFO: Longitudinal::configure() states.size()="
                       << (int)_states->size()
                       << "; meas.size()=" << (int)_measurements->size()
                       << "; act.size()=" << (int)_actuators->size() << Log::dp;

    // Save reference to actuator A6 for setting DL2 energy limits on every cycl
e.
    ActuatorSet::iterator it = _actuators->end();
    --it;
    _dl2EnergyActuator = (*it);

    // Make sure the correct devices names are in the right slot
    // in the measurements, actuators and states sets -> order is important!

    // Clear averages
    reset();

    // F and G matrices must have same dimensions
    if (_loopConfiguration->_fMatrix.size2() !=
            _loopConfiguration->_gMatrix.size2() ||
            _loopConfiguration->_fMatrix.size1() !=
            _loopConfiguration->_gMatrix.size1()) {
        _loopConfiguration->_logger << "Dimensions of F and G matrices not "
                << "the same, can't configure" << Log::flushpv;
        return -1;
    }

    // Change the DL2 gain in the G matrix according to the DL2 setpoint
    // The DL2 setpoint is given by the S6DES PV, which can be found
    // with the last item in the _states vector
    StateSet::iterator sIt = _states->end();
    --sIt;
    double dl2Setpoint = (*sIt)->getSetpoint();

    // Multiply F and G matrix (element-by-element)
    _fgMatrix = boost::numeric::ublas::element_prod(_loopConfiguration->_fMatrix
,
            _loopConfiguration->_gMatrix);

    // Reset IMAX running average (10 point sample)
    _loopConfiguration->_logger
            << "INFO: Loop configured with " << (int) _measurements->size()
            << " measurements, " << (int) _actuators->size()
```

```
                << " actuators, and " << (int) _states->size() << " states."
                << Log::flush;

    _pDelta.resize(LONGITUDINAL_MATRIX_DIMENSION);
    _iDelta.resize(LONGITUDINAL_MATRIX_DIMENSION);
    _delta.resize(LONGITUDINAL_MATRIX_DIMENSION);
    _latestMeasurements.resize(LONGITUDINAL_MATRIX_DIMENSION);
    _latestMeasurementsSum.resize(LONGITUDINAL_MATRIX_DIMENSION);
    _usedFgMatrix.resize(LONGITUDINAL_MATRIX_DIMENSION, LONGITUDINAL_MATRIX_DIME
NSION);

    for (int i = 0; i < LONGITUDINAL_MATRIX_DIMENSION; ++i) {
      _latestMeasurements[i] = 0;
      _latestMeasurementsSum[i] = 0;
      _delta[i] = 0;
    }

    for (int i = 0; i < MAX_LONGITUDINAL_MEASUREMENTS; ++i) {
        _actuatorFbckState[i] = false;
    }

    _firstCalculate = true;

    if (connectA6Limits() != 0) {
      _loopConfiguration->_logger << "ERROR: Failed to connect to A6 limits (using CA)"
                                  << Log::flush;
      return -1;
    }

    checkActuatorLimits();

    for (it = _actuators->begin(); it != _actuators->end(); ++it) {
      (*it)->clear(); // Reset pattern offsets
    }

    _previousMode = configuration->_mode;

    return 0;
}

/**
 * Create the CaChannel connections to the DL2 energy HIHI and LOLO
 * limits. If the automatic limit control (_dl2AutoLimitControl) is
 * enabled, try to connect to devices, otherwise do not connect.
 */
int Longitudinal::connectA6Limits() {
  // Connect to A6HIHI/LOLO local PV to automatically set DL2 energy limits
  if (_dl2EnergyActuator != NULL && _dl2AutoLimitControl) {
    std::string actuatorName = _dl2EnergyActuator->getName(); // name is FBCK:FB
xx:LG01:A6
    std::string limitName = "FBCK:FB04:LG01:" + actuatorName + "HIHI";
    if (_dl2HihiChannel == NULL) {
        _dl2HihiChannel = new CaChannel(CommunicationChannel::WRITE_ONLY, limitN
ame);
    }
    limitName = "FBCK:FB04:LG01:" + actuatorName + "LOLO";
    if (_dl2LoloChannel == NULL) {
        _dl2LoloChannel = new CaChannel(CommunicationChannel::READ_ONLY, limitNa
me);
    }
  }
  else {
    // Cannot control limit if actuator device is not present
    _dl2AutoLimitControl = false;
  }
  return 0;
}

/**
 * Clears all averages kept by the algorithm, set the SxxUSEDBYLOOP PVs
 * to false.
 *
 * @return always 0
```



```
 * @author L.Piccoli
 */
int Longitudinal::reset() {
    for (int i = 0; i < INTEGRAL_ENERGY_ELEMENTS; ++i) {
        _energyAverage[i]->clear();
    }

    for (int i = 0; i < INTEGRAL_CURRENT_ELEMENTS; ++i) {
        _currentAverage[i]->clear();
    }

    _blenBc1Average.clear();
    _blenBc2Average.clear();

    for (StateSet::iterator sIt = _states->begin();
         sIt != _states->end(); ++sIt) {
      (*sIt)->setUsedBy(false);
    }

    _previousMode = _loopConfiguration->_mode;

    return 0;
}

/**
 * This method overrides the Algorithm::setFeedbackDevices(). It sets the
 * actuator USEDBYLOOP PV according to the value of the states USED PV.
 * The setting of the FBCK PV for the Longitudital loop depends whether
 * the operator has selected or not the states.
 *
 * This is only called when there is a feedback mode change in the LoopThread or
 * when devices are disconnected (STATE PV is turned off).
 *
 * Actuator are ALL turned off if the actuators parameter is false. If actuators
 * is true, then only those actuators that have the equivalent state
 * used are turned on.
 *
 * @param measurements true indicates measurement FBCK PVs should be turned ON,
 * if false FBCK PV is turned OFF
 * @param actuators true indicates actuator FBCK PVs should be turned ON,
 * if false FBCK PV is turned OFF
 * @return 0 if all PVs were set, -1 if there was an error setting one PV.
 * @author L.Piccoli
 */
int Longitudinal::setFeedbackDevices(bool measurements, bool actuators) {
  Log::getInstance() << Log::flagFbckPv << Log::dpInfo
                     << "Longitudinal::setFeedbackDevices(";
  if (measurements) {
    Log::getInstance() << Log::flagFbckPv << "ON";
  }
  else {
    Log::getInstance() << Log::flagFbckPv << "OFF";
  }
  if (actuators) {
    Log::getInstance() << Log::flagFbckPv << ", ON)";
  }
  else {
    Log::getInstance() << Log::flagFbckPv << ", OFF)";
  }
  Log::getInstance() << Log::dp;

  StateSet::iterator sIt = _states->begin();
  ActuatorSet::iterator aIt = _actuators->begin();
  for (; aIt != _actuators->end(); ++aIt, ++sIt) {
      if (sIt != _states->end()) {
        // If actuators are being turned off it means the feedback is stopping
        // or getting into COMPUTE mode. This requires FBCK PVs to be
        // released
        if (!actuators) {
          (*aIt)->setUsedBy(false);
        }
        // If loop is being turned on or being ENABLED then only those
        // actuators whose equivalent state are used have the FBCK PV turned on.
```

```
280           else {
                (*alt)->setUsedBy((*sIt)->isUsed());
              }
            }
        }
285
        // If FBCK devices are being turned OFF, set SxxUSEDBYLOOP to false
        if (!actuators) {
          for (StateSet::iterator sIt = _states->begin();
               sIt != _states->end(); ++sIt) {
290         (*sIt)->setUsedBy(false);
          }
        }

        int value = Algorithm::setFeedbackDevices(measurements, actuators);
295
        // Wait for PAU RF to get things straight before starting
        Log::getInstance() << Log::flagFbckPv << Log::dpInfo
                           << "Longitudinal::setFeedbackDevices() sleeping 1 second..."
                           << Log::dp;
300     epicsThreadSleep(1);

        return value;
    }

305 /**
     * Invokes the calculate() with no parameters. All parameters passed
     * to this method have already been passed through configure().
     *
     * @author L.Piccoli
310  */
    int Longitudinal::calculate(LoopConfiguration &configuration,
            MeasurementSet &measurements,
            ActuatorSet &actuators,
            StateSet &states) throw (Exception) {
315     return calculate();
    };

    /**
     * Implementation of the Longitudinal algorithm.
320  *
     *
     * @return 0 on success, -1 on failure
     * @author L.Piccoli
     */
325 int Longitudinal::calculate() throw (Exception) {
      // Save state of actuators FBCK PV on the first call to calculate()
      if (_firstCalculate) {
        int i = 0;
        for (ActuatorSet::iterator it = _actuators->begin();
330          it != _actuators->end() && i < MAX_LONGITUDINAL_MEASUREMENTS;
             ++it, ++i) {
          _actuatorFbckState[i] = (*it)->getFeedbackPv();
        }
        _firstCalculate = false;
335   }

      // First verify current energy and calculate actuator limits.
      checkActuatorLimits();

340   // Gather current measurements and select state based on readings
      // and USED PVs.
      if (checkConfiguration() != 0) {
          return -1;
      }
345
      // Zero out rows/cols of those states not in use
      _configureMatrixStats.start();
      configureMatrix();
      _configureMatrixStats.end();
350
      _calculateEnergyCurrentStats.start();
      if (calculateEnergiesAndCurrents() != 0) {
```



```
        _calculateEnergyCurrentStats.end();
          return -1;
355   }
      _calculateEnergyCurrentStats.end();

      // Calculate states based on the energies and currents
      _updateStatesStats.start();
360   if (updateStates() != 0) {
        _updateStatesStats.end();
        return STATE_LIMIT;
      }
      _updateStatesStats.end();
365
      // Calculate new actuator values
      _updateActuatorsStats.start();
      if (updateActuators() != 0) {
        _updateActuatorsStats.end();
370       return ACTUATOR_LIMIT;
      }
      _updateActuatorsStats.end();

      return 0;
375 }

    /**
     * Verify the current energy and set the correct limits for L3, if the
     * _dl2AutoLimitControl is set to true. Otherwise L3 limits can be
380  * set through the Longitudinal EDM panel.
     *
     * @author L.Piccoli
     */
    void Longitudinal::checkActuatorLimits() {
385   if (_dl2AutoLimitControl) {
        double loloLimit = ExecConfiguration::getInstance()._dl2EnLoloPv.getValue();
        double hihiLimit = ExecConfiguration::getInstance()._dl2EnHihiPv.getValue();

        // TODO: Does this propagates to the PV immediately?
390     // this new value must be available on this iteration
        if (_dl2EnergyActuator->getHihi() != hihiLimit) {
          _dl2HihiChannel->write(hihiLimit);
        }
        if (_dl2EnergyActuator->getLolo() != loloLimit) {
395       _dl2LoloChannel->write(loloLimit);
        }
      }
    }

400 /**
     * Collect current measurements and select states based on the readings and
     * user configuration (SxxUSED PVs)
     *
     * @return 0 on success, -1 on failure
405  * @author L.Piccoli
     */
    int Longitudinal::checkConfiguration() throw (Exception) {
        // Check which measurements are valid
        _checkMeasurementsStats.start();
410     if (checkMeasurementStatus() != 0) {
          _checkMeasurementsStats.end();
            return -1;
        }
        _checkMeasurementsStats.end();
415
        // Check the USEDBY PV for the states based on the measurement status
        // and SxxUSED PVs
        _selectStatesStats.start();
        if (selectStates() != 0) {
420       _selectStatesStats.end();
            return -1;
        }
        _selectStatesStats.end();

425     // Check if beam has changed direction and clean averages if so.
```

```
        setPaths();

        return 0;
    }

    /**
     * Generate an F.*G matrix with rows/cols zeroed for those states not in
     * use for the current cycle.
     *
     * @author L.Piccoli
     */
    void Longitudinal::configureMatrix() {
      // TODO: If the gains are to be changed according to the DL2 setpoint,
      // then the _fgMatrix will have to be calculated in this method.
      // Currently it is calculated when the algorithm is configured.
      // An option is to have several _fgMatrices, one for each energy...

        StateSet::iterator stateRowIt = _states->begin();
        for (int i = 0; i < (int) _fgMatrix.size1(); ++i, ++stateRowIt) {
            StateDevice *rowState = (*stateRowIt);
            StateSet::iterator stateColIt = _states->begin();
            for (int j = 0; j < (int) _fgMatrix.size2(); ++j, ++stateColIt) {
                StateDevice *colState = (*stateColIt);
                if (!rowState->getUsedBy()) {
                    _usedFgMatrix(i, j) = 0;
                } else if (!colState->getUsedBy()) {
                    _usedFgMatrix(i, j) = 0;
                } else {
                   _usedFgMatrix(i, j) = _fgMatrix(i, j);
                }
            }
        }
    }

    /**
     * Calculate the energies and currents at DL1, BC1, BC2 and DL2.
     *
     * Energies and currents are stored in the _energyAverage and _currentAverage
     * attributes.
     *
     * @return always 0
     * @author L.Piccoli
     */
    int Longitudinal::calculateEnergiesAndCurrents() {
        StateSet::iterator stateIt = _states->begin();
        MeasurementSet::iterator measurementIt = _measurements->begin();

        // Energy at DL1
        MeasurementDevice *m1 = *measurementIt;
        ++measurementIt; // Go to M2
        MeasurementDevice *m2 = *measurementIt;
        double res = calculateEnergyDl1(m1, m2, *stateIt);
        if (std::isnan(res)) {
          return -1;
        }

        // Energy at BC1
        ++measurementIt; // Go to M3
        ++stateIt; // S2
        res = calculateEnergyBc1(*measurementIt, *stateIt);
        if (std::isnan(res)) {
          return -1;
        }

        // Current at BC1
        ++measurementIt; // Go to M4, it's a BLEN
        ++stateIt; // Go to S3
        res = calculateCurrentBc1(*stateIt);
        if (std::isnan(res)) {
          return -1;
        }

        // Energy at BC2
```



```
        ++measurementIt; // Go to M5
        ++stateIt; // Go to S4
        res = calculateEnergyBc2(*measurementIt, *stateIt);
        if (std::isnan(res)) {
          return -1;
        }

        // Current at BC2
        ++measurementIt; // Go to M6, it's a BLEN
        ++stateIt; // Go to S5
        res = calculateCurrentBc2(*stateIt);
        if (std::isnan(res)) {
          return -1;
        }

        // Energy at DL2
        ++measurementIt; // Go to M7
        MeasurementDevice *m7 = *measurementIt;
        ++measurementIt; // Go to M8
        MeasurementDevice *m8 = *measurementIt;
        ++measurementIt; // Go to M9
        MeasurementDevice *m9 = *measurementIt;
        ++measurementIt; // Go to M10
        MeasurementDevice *m10 = *measurementIt;
        ++stateIt; // Go to S6
        res = calculateEnergyDl2(m7, m8, m9, m10, *stateIt);
        if (std::isnan(res)) {
          return -1;
        }

        return 0;
    }

    /**
     * Calculate the energy at DL1. Beam must be either going through M1 or M2.
     * This method should not be called if there is no beam.
     *
     * @param m1 BPMS:IN20:731
     * @param m2 BPMS:IN20:981
     * @param s1 State at DL1, used to retrieve the S1 setpoint
     * @return energy at DL1
     * @author L.Piccoli
     */
    double Longitudinal::calculateEnergyDl1(MeasurementDevice *m1, MeasurementDevice
     *m2,
            StateDevice *s1) {
        if (!s1->getUsedBy()) {
            _latestMeasurements[0] = 0;
            _latestMeasurementsSum[0] = 0;
            return 0;
        }

        double dispersion = 0;
        MeasurementDevice *measurement = m1;

        // Do we have beam on M1?
        if (_measurementStatus[M1] == VALID) {
            dispersion = _loopConfiguration->_dispersions[M1];
        }// If not on M1 it must be going through M2
        else {
            measurement = m2;
            dispersion = _loopConfiguration->_dispersions[M2];
        }
        /*
        double energy = (measurement->peek() / dispersion + 1) *
                (135.0 / s1->getSetpoint()) - 1;
        */
        // Joe's algorithm
        //    double energy = measurement->peek() * (s1->getSetpoint()/dispersion);
        double refEnergy = ExecConfiguration::getInstance()._dl1ErefPv.getValue() *
    1000;
        double energy = measurement->peek() * (refEnergy/dispersion) + refEnergy - s
    1->getSetpoint();
```

```
570     _latestMeasurements[0] = energy;
        _energyAverage[DL1_ENERGY]->add(energy);
        _latestMeasurementsSum[0] = _energyAverage[DL1_ENERGY]->getSum();
        if (std::isnan(_latestMeasurementsSum[0])) {

575       _loopConfiguration->_logger << Log::flagAlgo << Log::dpError
                                    << "ERROR: DL1 - std::isnan(_latestMeasurementsSum[0]) (head
="
                                    << (int)_energyAverage[DL1_ENERGY]->getHead()
<< ", sum="
                                    << (double)_energyAverage[DL1_ENERGY]->getSum(
) << ", M1/2 = "
                                    << (double)measurement->peek() << ", energy="
580                                 << (double)energy << ")" << Log::dp;
            return _latestMeasurementsSum[0];
        }

        return energy;
585 }

    /**
     * Calculate the energy at BC1. Beam must be going through BC1.
     *
590  * The calculated energy is added to the _energyAverage[BC1_ENERGY], which
     * mainains an average of the last 3 calculated energies.
     *
     * @param m3 BPMS:LI21:233
     * @param s2 State at BC1 energy, used to retrieve S2 setpoint
595  * @return energy at BC1
     * @author L.Piccoli
     */
    double Longitudinal::calculateEnergyBc1(MeasurementDevice* m3, StateDevice* s2)
    {
        if (!s2->getUsedBy()) {
600         _latestMeasurements[1] = 0;
            _latestMeasurementsSum[1] = 0;
        }
        /*
        double energy = (m3->peek() / _loopConfiguration->_dispersions[M3] + 1) *
605           (220.0 / s2->getSetpoint()) - 1;
        */
        // Joe's algorithm
        //    double energy = m3->peek() * (s2->getSetpoint()/_loopConfiguration->_d
ispersions[M3]);
        double refEnergy = ExecConfiguration::getInstance()._bc1ErefPv.getValue() *
1000;
610     double energy = m3->peek() * (refEnergy/_loopConfiguration->_dispersions[M3]
) +
          refEnergy - s2->getSetpoint();

        _latestMeasurements[1] = energy;
        _energyAverage[BC1_ENERGY]->add(energy);
615     _latestMeasurementsSum[1] = _energyAverage[BC1_ENERGY]->getSum();
        if (std::isnan(_latestMeasurementsSum[1])) {

          _loopConfiguration->_logger << Log::flagAlgo << Log::dpError
                                      << "ERROR: BC1 Energy - std::isnan(_latestMeasurementsSum[1
])"
620                                   << " (head=" << (int)_energyAverage[BC1_ENERGY]
->getHead()
                                      << ", sum=" << (double)_energyAverage[BC1_ENERG
Y]->getSum()
                                      << ", M3 = " << (double) m3->peek()
                                      << ", energy=" << (double) energy << ")" << Log::
dp;

625         return _latestMeasurementsSum[1];
        }

        return energy;
    }
630
```



```
    /**
     * Calculate the current at BC1 based on the average of the last 10 BLEN reading
s.
     * The average is retrieved from the _blenBc1Average attribute, that's why there
     * is no MeasurementDevice* parameter as in the calculateEnergy*() methods.
635  *
     * The calculated current is added to the _currentAverage[BC1_CURRENT], which
     * mainains an average of the last 3 calculated energies.
     *
     * @param s3 State at BC1 current, used to retrieve S3 setpoint
640  * @return current at BC1
     * @author L.Piccoli
     */
    double Longitudinal::calculateCurrentBc1(StateDevice *s3) {
        if (!s3->getUsedBy()) {
645         _latestMeasurements[2] = 0;
            _latestMeasurementsSum[2] = 0;
        }

        double sign = 0;
650     double setpoint = s3->getSetpoint();
        if (setpoint > 0) {
            sign = 1;
        } else if (setpoint < 0) {
            sign = -1;
655     }
        double current = (_blenBc1Average.getAverage() - std::abs(setpoint)) /
                  std::abs(setpoint);
        current *= sign;

660     _latestMeasurements[2] = current;
        _currentAverage[BC1_CURRENT]->add(current);
        _latestMeasurementsSum[2] = _currentAverage[BC1_CURRENT]->getSum();
        if (std::isnan(_latestMeasurementsSum[2])) {

665       _loopConfiguration->_logger << Log::flagAlgo << Log::dpError
                                      << "ERROR: BC1 Cur - std::isnan(_latestMeasurementsSum[2])"
                                      << " (head=" << (int)_currentAverage[BC1_CURREN
T]->getHead()
                                      << ", sum=" << (double)_currentAverage[BC1_CURR
ENT]->getSum()
                                      << ", M4=" << (double)_blenBc1Average.getLates
tValue()
670                                   << ", M4AVG=" << (double)_blenBc1Average.getAv
erage()
                                      << ", current=" << current << ")" << Log::dp;

          return _latestMeasurementsSum[2];
        }
675
        return current;
    }

    /**
680  * Calculate the energy at BC2. Beam must be going through BC2.
     *
     * The calculated energy is added to the _energyAverage[BC2_ENERGY], which
     * maintains an average of the last 3 calculated energies.
     *
685  * @param m5 BPMS:LI24:801
     * @param s4 State at BC1 energy, used to retrieve S2 setpoint
     * @return energy at BC1
     * @author L.Piccoli
     */
690 double Longitudinal::calculateEnergyBc2(MeasurementDevice* m5, StateDevice* s4)
    {
        if (!s4->getUsedBy()) {
            _latestMeasurements[3] = 0;
            _latestMeasurementsSum[3] = 0;
        }
695     /*
        double energy = (m5->peek() / _loopConfiguration->_dispersions[M5] + 1) *
          (5000.0 / s4->getSetpoint()) - 1;
```

```
        */
        // Joe's algorithm
    700 //    double energy = m5->peek() * (s4->getSetpoint()/_loopConfiguration->_d
    ispersions[M5]);
        double refEnergy = ExecConfiguration::getInstance()._bc2ErefPv.getValue() *
    1000;
        double energy = m5->peek() * (refEnergy/_loopConfiguration->_dispersions[M5]
    ) +
          refEnergy - s4->getSetpoint();

    705 _latestMeasurements[3] = energy;
        _energyAverage[BC2_ENERGY]->add(energy);
        _latestMeasurementsSum[3] = _energyAverage[BC2_ENERGY]->getSum();
        if (std::isnan(_latestMeasurementsSum[3])) {
          _loopConfiguration->_logger << Log::flagAlgo << Log::dpError
    710                               << "ERROR: BC2 Energy - std::isnan(_latestMeasurementsSum[3
    ])"
                                      << " (head=" << (int)_energyAverage[BC2_ENERGY]
    ->getHead()
                                      << ", sum=" << (double)_energyAverage[BC2_ENERG
    Y]->getSum()
                                      << ", M5 = " << (double)m5->peek()
                                      << ", energy=" << (double)energy << ")" << Log::d
    p;
    715     return _latestMeasurementsSum[3];
        }

        return energy;
    }
    720
    /**
     * Calculate the current at BC2 based on the average of the last 10 BLEN reading
    s.
     * The average is retrieved from the _blenBc2Average attribute, that's why there
     * is no MeasurementDevice* parameter as in the calculateEnergy*() methods.
    725 *
     * The calculated current is added to the _currentAverage[BC2_CURRENT], which
     * mainains an average of the last 3 calculated energies.
     *
     * @param s5 State at BC2 current, used to retrieve S3 setpoint
    730 * @return current at BC2
     * @author L.Piccoli
     */
    double Longitudinal::calculateCurrentBc2(StateDevice *s5) {
        if (!s5->getUsedBy()) {
    735     _latestMeasurements[4] = 0;
            _latestMeasurementsSum[4] = 0;
        }

        double sign = 0;
    740 double setpoint = s5->getSetpoint();
        if (setpoint > 0) {
            sign = 1;
        } else if (setpoint < 0) {
            sign = -1;
    745 }
        double current = (_blenBc2Average.getAverage() - std::abs(setpoint)) /
                std::abs(setpoint);
        current *= sign;

    750 _latestMeasurements[4] = current;
        _currentAverage[BC2_CURRENT]->add(current);
        _latestMeasurementsSum[4] = _currentAverage[BC2_CURRENT]->getSum();
        if (std::isnan(_latestMeasurementsSum[4])) {
          _loopConfiguration->_logger << Log::flagAlgo << Log::dpError
    755                               << "ERROR: BC2 Curr - std::isnan(_latestMeasurementsSum[4])
    "
                                      << " (head=" << (int)_currentAverage[BC2_CURREN
    T]->getHead()
                                      << ", sum=" << (double)_currentAverage[BC2_CURR
    ENT]->getSum()
                                      << ", M6=" << (double)_blenBc2Average.getLates
    tValue()
```



```
                                      << ", M6AVG=" << (double)_blenBc2Average.getAv
    erage()
    760                               << ", current=" << (double)current << ")" << Log:
    :dp;

            return _latestMeasurementsSum[4];
        }

    765 return current;
    }

    /**
     * Calculate the energy at DL2. Beam must be either to BSY, BYKIK or
    770 * LTU.
     *
     * @param m7 BPMS:BSY0:52
     * @param m8 BPMS:LTU1:250
     * @param m9 BPMS:LTU1:450
    775 * @param m10 BPMS:LTU0:170
     * @param s6 State at DL2 energy, used to retrieve S6 setpoint
     * @return energy at DL2
     * @author L.Piccoli
     */
    780 double Longitudinal::calculateEnergyDl2(MeasurementDevice* m7,
            MeasurementDevice* m8, MeasurementDevice* m9,
            MeasurementDevice* m10, StateDevice* s6) {
        if (!s6->getUsedBy()) {
            _latestMeasurements[5] = 0;
    785     _latestMeasurementsSum[5] = 0;
        }

        double energy = 0;
        //    double by1Bdes = ExecConfiguration::getInstance()._by1BdesPv.getValue(
    );
    790 //    by1Bdes *= 1000;
        double refEnergy = ExecConfiguration::getInstance()._dl2ErefPv.getValue() *
    1000;

        // Beam is going through BSY
        if (_measurementStatus[M7] == VALID) {
    795   /*
            energy = (m7->peek() / _loopConfiguration->_dispersions[M7] + 1) *
                    (by1Bdes / s6->getSetpoint()) - 1;
          */
            // Joe's algorithm
    800   //      energy = m7->peek() * (s6->getSetpoint()/_loopConfiguration->_di
    spersions[M7]);
            energy = m7->peek() * (refEnergy/_loopConfiguration->_dispersions[M7]) +
              refEnergy - s6->getSetpoint();
        } else {
            // Energy problem, trying to recover with M10
    805     if (_measurementStatus[M10] == M10ENERGY) {
              /*
                energy = (m10->peek() / _loopConfiguration->_dispersions[M10] + 1) *
                        (by1Bdes / s6->getSetpoint()) - 1;
              */
    810       // Joe's algorithm
              //      energy = m10->peek() * (s6->getSetpoint()/_loopConfiguration
    ->_dispersions[M10]);
              energy = m10->peek() * (refEnergy/_loopConfiguration->_dispersions[M10
    ]) +
                refEnergy - s6->getSetpoint();
            } else {
    815       // Beam is going through BYKIK
              if (_measurementStatus[M8] == VALID &&
                      _measurementStatus[M9] == INVALID) {
                /*
                  energy = (m8->peek() / _loopConfiguration->_dispersions[M8] + 1)
     *
    820                     (by1Bdes / s6->getSetpoint()) - 1;
                */
                // Joe's algorithm
                // energy = m8->peek() * (s6->getSetpoint()/_loopConfiguration->_d
```

```
ispersions[M8]);
                energy = m8->peek() * (refEnergy/_loopConfiguration->_dispersions[
M8]) +
                  refEnergy - s6->getSetpoint();
              } else { // Beam is going through LTU
                // Original
                /*
                  energy = m8->peek() / _loopConfiguration->_dispersions[M8];
                  energy += m9->peek() / _loopConfiguration->_dispersions[M9];
                  energy /= 2;
                  energy += 1;
                  energy *= (by1Bdes / s6->getSetpoint());
                  energy -= 1;
                  energy *= s6->getSetpoint();
                  */

                // Joe's algorithm (1)
                energy = ((((m8->peek()/_loopConfiguration->_dispersions[M8])+
                            (m9->peek()/_loopConfiguration->_dispersions[M9]) ) /2
 ) * refEnergy) +
                  refEnergy - s6->getSetpoint();
              }
          }
      }

      _latestMeasurements[5] = energy;
      _energyAverage[DL2_ENERGY]->add(energy);
      _latestMeasurementsSum[5] = _energyAverage[DL2_ENERGY]->getSum();
      if (std::isnan(_latestMeasurementsSum[5])) {

        _loopConfiguration->_logger << Log::flagAlgo << Log::dpError
                                    << "ERROR: DL2 - std::isnan(_latestMeasurementsSum[5])"
                                    << " (head=" << (int)_energyAverage[DL2_ENERGY]
->getHead()
                                    << ", sum=" << (double)_energyAverage[DL2_ENERG
Y]->getSum()
                                    << ", M7=" << (double)m7->peek()
                                    << ", M8=" << (double)m8->peek()
                                    << ", M9=" << (double)m9->peek()
                                    << ", M10=" << (double)m10->peek()
                                    << ", energy=" << (double)energy << ")" << Log::d
p;

          return _latestMeasurementsSum[5];
      }

      return energy;
}

/**
 * Calculate states for DL1, BC1, BC2 and DL2 based on the latest calculated
 * energies and currents.
 *
 * If a state is not used a new value is not calculated.
 *
 * @return always 0
 * @author L.Piccoli
 */
int Longitudinal::updateStates() throw (Exception) {
      StateSet::iterator stateIt = _states->begin();
      bool stateLimit = false;
      //    std::string exceptionMessage = "State(s) out of range:";
      unsigned int stateLimitFlag = 0;

      // Energy at DL1
      if ((*stateIt)->isUsed()) {
        if (calculateEnergyState(*stateIt, _energyAverage[DL1_ENERGY]->getLatestVa
lue()) != 0) {
          stateLimit = true;
          stateLimitFlag |= 0x01;
        }
      }
```



```
      // Energy at BC1
      ++stateIt;
      if ((*stateIt)->isUsed()) {
        if (calculateEnergyState(*stateIt, _energyAverage[BC1_ENERGY]->getLatestVa
lue()) != 0) {
          stateLimit = true;
          stateLimitFlag |= 0x02;
        }
      }

      // Current at BC1
      ++stateIt;
      if ((*stateIt)->isUsed()) {
        if (calculateCurrentState(*stateIt, _currentAverage[BC1_CURRENT]->getLates
tValue()) != 0) {
          stateLimit = true;
          stateLimitFlag |= 0x04;
        }
      }

      // Energy at BC2
      ++stateIt;
      if ((*stateIt)->isUsed()) {
        if (calculateEnergyState(*stateIt, _energyAverage[BC2_ENERGY]->getLatestVa
lue()) != 0) {
          stateLimit = true;
          stateLimitFlag |= 0x08;
        }
      }

      // Current at BC2
      ++stateIt;
      if ((*stateIt)->isUsed()) {
        if (calculateCurrentState(*stateIt, _currentAverage[BC2_CURRENT]->getLates
tValue()) != 0) {
          stateLimit = true;
          stateLimitFlag |= 0x10;
        }
      }

      // Energy at DL2
      ++stateIt;
      if ((*stateIt)->isUsed()) {
        if (calculateEnergyState(*stateIt, _energyAverage[DL2_ENERGY]->getLatestVa
lue()) != 0) {
          stateLimit = true;
          stateLimitFlag |= 0x20;
        }
      }

      if (stateLimitFlag > 0) {
        _exceptionMessage = "State(s) out of range:";
        stateIt = _states->begin();
        if (stateLimitFlag & 0x01) {
          _exceptionMessage += " ";
          _exceptionMessage += (*stateIt)->getDeviceName();
        }
        ++stateIt;
        if (stateLimitFlag & 0x02) {
          _exceptionMessage += ", ";
          _exceptionMessage += (*stateIt)->getDeviceName();
        }
        ++stateIt;
        if (stateLimitFlag & 0x04) {
          _exceptionMessage += ", ";
          _exceptionMessage += (*stateIt)->getDeviceName();
        }
        ++stateIt;
        if (stateLimitFlag & 0x08) {
          _exceptionMessage += ", ";
          _exceptionMessage += (*stateIt)->getDeviceName();
        }
        ++stateIt;
```

```
        if (stateLimitFlag & 0x10) {
          _exceptionMessage += ",";
          _exceptionMessage += (*stateIt)->getDeviceName();
        }
        ++stateIt;
        if (stateLimitFlag & 0x20) {
          _exceptionMessage += ",";
          _exceptionMessage += (*stateIt)->getDeviceName();
        }
        throw Exception(_exceptionMessage, Exception::STATE_LIMIT);
        return -1;
      }
      return 0;
  }

  /**
   * Calculate a new state value based on the latest current. The state
   * value is computed only if the state is in use, i.e. there is beam going
   * through the section and the user has selected the state.
   *
   * @param state pointer to the state device to be updated.
   * @param latestValue lastest energy or current
   * @return always 0
   * @author L.Piccoli
   */
  int Longitudinal::calculateCurrentState(StateDevice *state, double latestValue)
  {
      if (state->getUsedBy()) {
          double stateValue = 1 + latestValue;
          stateValue *= state->getSetpoint();
          if (state->set(stateValue) != 0) {
              return -1;
          }
      }
      return 0;
  }

  /**
   * Calculate a new state value based on the latest energy. The state
   * value is computed only if the state is in use, i.e. there is beam going
   * through the section and the user has selected the state.
   *
   * @param state pointer to the state device to be updated.
   * @param latestValue lastest energy or current
   * @return always 0
   * @author L.Piccoli
   */
  int Longitudinal::calculateEnergyState(StateDevice *state, double latestValue) {
      if (state->getUsedBy()) {
          double stateValue = state->getSetpoint() + latestValue;
          if (state->set(stateValue) != 0) {
              return -1;
          }
      }
      return 0;
  }

  /**
   * Verify which measurements are currently valid. The _measurementStatus
   * array is updated accordingly.
   *
   * A BPM measurement is VALID if TMIT is above the threshold (TMITLOW PV).
   *
   * A BLEN measurement is VALID if IMAX is greater than ZERO.
   *
   * These are all the measurement devices in the Longitudinal feedback:
   *
   *   M1 BPMS:IN20:731:X --\
   *   M2 BPMS:IN20:981:X ---+---> Beam goes through one BPM or the other
   *   M3 BPMS:LI21:233:X
   *   M4 BLEN:LI21:265:AIMAXF2
   *   M5 BPMS:LI24:801:X
   *   M6 BLEN:LI24:886:BIMAXF2
```



```
   *   M7 BPMS:BSY0:52:X
   *   M8 BPMS:LTU1:250:X ---+---> These two BPMs are combined
   *   M9 BPMS:LTU1:450:X --/
   *   M10 BPMS:LTU0:170:Y -> Used when there are problems setting up the energy
   *
   * Additionally, this method keeps track on whether the beam is going
   * through different beam lines at DL1 and BSY. If there is a change
   * then the DL1 and DL2 integral energies must be cleared out.
   *
   * @return 0 if at least one measurement device is valid, -1 if all have
   * invalid measurements
   * @author L.Piccoli
   */
  int Longitudinal::checkMeasurementStatus() {
      MeasurementDevice *measurementDevice;
      double value;
      epicsTimeStamp timestamp;
      int i = 0;
      bool allInvalid = true;

      for (MeasurementSet::iterator it = _measurements->begin();
              it != _measurements->end() && i < MAX_LONGITUDINAL_MEASUREMENTS;
              ++it, ++i) {
          measurementDevice = (*it);
          measurementDevice->get(value, timestamp);
          _measurementStatus[i] = INVALID;

          if (measurementDevice->isBpm()) {
              if (measurementDevice->getTmit() >
                      ExecConfiguration::getInstance()._tmitLowPv.getValue()) {
                  _measurementStatus[i] = VALID;
                  allInvalid = false;
              }
          } else if (measurementDevice->isBlen()) {
              if (value > 0 && !std::isnan(value)) {
                  _measurementStatus[i] = VALID;
                  allInvalid = false;
                  // Copy latest BLEN values to the appropriate average holder
                  if (i == M4) {
                      _blenBc1Average.add(value);
                  } else if (i == M6) {
                      _blenBc2Average.add(value);
                  }
              }
          }
          if (PatternManager::getInstance().getCurrentPattern().isTs4()) {
            measurementDevice->setUsedBy(_measurementStatus[i] == VALID);
          }
      }

      if (allInvalid) {
          _loopConfiguration->_logger << "No valid measurements, no beam"
                  << Log::flushpvonly;
          return -1;
      } else {
          return 0;
      }
  }

  /**
   * Select the states to be used in the current cycle based on the status
   * of BPM/BLEN measurements. If a state is used the USEDBY PV is set to true.
   *
   * These are the state devices:
   *
   *   S1 DL1.Energy - USEDBY defined by S1USED & M1 TMIT or M2 TMIT
   *   S2 BC1.Energy - USEDBY defined by S2USED & M3 TMIT
   *   S3 BC1.Current - USEDBY defined by S3USED & M4 IMAX
   *   S4 BC2.Energy - USEDBY defined by S4USED & M5 TMIT
   *   S5 BC2.Current - USEDBY defined by S5USED & M6 IMAX
   *   S6 DL2.Energy - USEDBY defined by S6USED & M7-M8/M9/M10 TMITs
   *
   * If a state USED PV and the actuator FBCK PV have different values
```

```
     * the state of the FBCK PV changes to the value given by the state USED PV.
     * This allows actuators to be released from the feedback control once the
1105 * operators disable the equivalent state calculation.
     *
     * @return 0 on success, -1 if the measurements status do not reflect a valid
     * configuration (e.g. TMIT of M7 and M8 are above TMITLOW PV)
     * @author L.Piccoli
1110 */
    int Longitudinal::selectStates() throw (Exception) {
        setActuatorFeedbackPvs();

        StateSet::iterator it = _states->begin();
1115    StateDevice *stateDevice;
        ActuatorSet::iterator actIt = _actuators->begin();
        ActuatorDevice *actuatorDevice;
        MeasurementSet::iterator measIt = _measurements->begin();
        MeasurementDevice *measurementDevice;
1120    bool stateUsed = false;
        bool hasState = false;
        bool currentMode = _loopConfiguration->_mode;

        // M1 and M2 define whether S1 is enabled
1125    if (it == _states->end()) {
          throw Exception("Cannot access DL1 energy state, check feedback configuration, S1USED must be 1"
    );
        }
        if (actIt == _actuators->end()) {
          throw Exception("Cannot access A1, check feedback configuration");
1130    }

        stateDevice = (*it);
        actuatorDevice = (*actIt);
        if ((_measurementStatus[M1] == VALID ||
1135          _measurementStatus[M2] == VALID) &&
             stateDevice->isUsed()) {
           stateUsed = true;
           hasState = true;
           measurementDevice = (*measIt); // M1
1140       if (_measurementStatus[M1] == VALID) {
             measurementDevice->setUsedBy(true);
           }
           else {
             measurementDevice->setUsedBy(false);
1145       }
           ++measIt; // M2
           measurementDevice = (*measIt);
           if (_measurementStatus[M2] == VALID) {
             measurementDevice->setUsedBy(true);
1150       }
           else {
             measurementDevice->setUsedBy(false);
           }
        }
1155    // Get initial actuator value if state was just turned on
        // and feedback is in the ENABLE mode.
        if ((stateDevice->getUsedBy() == false &&
             stateDevice->isUsed() &&
             stateUsed == true &&
1160         currentMode == true) ||
            (_previousMode == false &&
             currentMode == true &&
             stateDevice->isUsed())) {
          Log::getInstance() << Log::flagFbckPv << Log::dpInfo
1165                         << "Longitudinal::selectStates() "
                             << "reading initial values for "
                             << actuatorDevice->getDeviceName().c_str()
                             << Log::dp;
          if (actuatorDevice->getInitialSetting() != 0) {
1170        throw Exception("Failed to get initial value for A1");
          }
          // Clear averages related to DL1
          _energyAverage[DL1_ENERGY]->clear();
        }
```



```
1175    stateDevice->setUsedBy(stateUsed);
        actuatorDevice->setUsedBy(stateUsed);
        if (!stateUsed) {
          double value = actuatorDevice->getLastValueSet();
          actuatorDevice->set(value, DataPoint::SET_SKIP);
1180    }

        // M3 defines if S2 is enabled
        ++it;
        stateDevice = (*it);
1185    ++actIt;
        actuatorDevice = (*actIt);
        if (it == _states->end()) {
          throw Exception("Cannot access BC1 energy state, check feedback configuration, S2USED must be 1"
    );
        }
1190    if (actIt == _actuators->end()) {
          throw Exception("Cannot access A2, check feedback configuration");
        }
        stateUsed = false;
        if (_measurementStatus[M3] == VALID &&
1195          stateDevice->isUsed()) {
           stateUsed = true;
           hasState = true;
           ++measIt; // M3
           measurementDevice = (*measIt);
1200       if (_measurementStatus[M3] == VALID) {
             measurementDevice->setUsedBy(true);
           }
           else {
             measurementDevice->setUsedBy(false);
1205       }
        }
        // Get initial actuator value if state was just turned on
        // and feedback is in the ENABLE mode.
        if ((stateDevice->getUsedBy() == false &&
1210         stateDevice->isUsed() &&
             stateUsed == true &&
             currentMode == true) ||
            (_previousMode == false &&
             currentMode == true &&
1215         stateDevice->isUsed())) {
          Log::getInstance() << Log::flagFbckPv << Log::dpInfo
                             << "Longitudinal::selectStates() "
                             << "reading initial values for "
                             << actuatorDevice->getDeviceName().c_str()
1220                         << Log::dp;
          if (actuatorDevice->getInitialSetting() != 0) {
            throw Exception("Failed to get initial value for A2");
          }
          // Clear averages related to BC1 energy
1225      _energyAverage[BC1_ENERGY]->clear();
        }
        stateDevice->setUsedBy(stateUsed);
        actuatorDevice->setUsedBy(stateUsed);
        if (!stateUsed) {
1230      double value = actuatorDevice->getLastValueSet();
          actuatorDevice->set(value, DataPoint::SET_SKIP);
        }

        // M3 also defines if S3 is enabled
1235    // We cannot rely on M4 BLEN measurement
        ++it;
        stateDevice = (*it);
        ++actIt;
        actuatorDevice = (*actIt);
1240    if (it == _states->end()) {
          throw Exception("Cannot access S3, check feedback configuration");
        }
        if (actIt == _actuators->end()) {
          throw Exception("Cannot access A3, check feedback configuration");
1245    }
```

```
        stateUsed = false;
        if (_measurementStatus[M3] == VALID &&
                stateDevice->isUsed()) {
1250        stateUsed = true;
            hasState = true;
            ++measIt; // M4
            measurementDevice = (*measIt);
            if (_measurementStatus[M3] == VALID) {
1255          measurementDevice->setUsedBy(true);
            }
            else {
              measurementDevice->setUsedBy(false);
            }
1260    }
        // Get initial actuator value if state was just turned on
        // and feedback is in the ENABLE mode.
        if ((stateDevice->getUsedBy() == false &&
             stateDevice->isUsed() &&
1265         stateUsed == true &&
             currentMode == true) ||
            (_previousMode == false &&
             currentMode == true &&
             stateDevice->isUsed())) {
1270      Log::getInstance() << Log::flagFbckPv << Log::dpInfo
                             << "Longitudinal::selectStates() "
                             << "reading initial values for "
                             << actuatorDevice->getDeviceName().c_str()
                             << Log::dp;
1275      if (actuatorDevice->getInitialSetting() != 0) {
            throw Exception("Failed to get initial value for A3");
          }
          // Clear averages related to BC1 current
          _currentAverage[BC1_CURRENT]->clear();
1280      _blenBc1Average.clear();
        }
        stateDevice->setUsedBy(stateUsed);
        actuatorDevice->setUsedBy(stateUsed);
        if (!stateUsed) {
1285      double value = actuatorDevice->getLastValueSet();
          actuatorDevice->set(value, DataPoint::SET_SKIP);
        }

        // M5 defines if S4 is enabled
1290    ++it;
        stateDevice = (*it);
        ++actIt;
        actuatorDevice = (*actIt);
        if (it == _states->end()) {
1295      throw Exception("Cannot access S4, check feedback configuration");
        }
        if (actIt == _actuators->end()) {
          throw Exception("Cannot access A4, check feedback configuration");
        }
1300    stateUsed = false;
        if (_measurementStatus[M5] == VALID &&
                stateDevice->isUsed()) {
            stateUsed = true;
            hasState = true;
1305        ++measIt; // M5
            measurementDevice = (*measIt);
            if (_measurementStatus[M5] == VALID) {
              measurementDevice->setUsedBy(true);
            }
1310        else {
              measurementDevice->setUsedBy(false);
            }
        }
        // Get initial actuator value if state was just turned on
1315    // and feedback is in the ENABLE mode.
        if ((stateDevice->getUsedBy() == false &&
             stateDevice->isUsed() &&
             stateUsed == true &&
             currentMode == true) ||
```



```
1320        (_previousMode == false &&
             currentMode == true &&
             stateDevice->isUsed())) {
          Log::getInstance() << Log::flagFbckPv << Log::dpInfo
                             << "Longitudinal::selectStates() "
1325                         << "reading initial values for "
                             << actuatorDevice->getDeviceName().c_str()
                             << Log::dp;
          if (actuatorDevice->getInitialSetting() != 0) {
            throw Exception("Failed to get initial value for A4");
1330      }
          // Clear averages related to BC2 energy
          _energyAverage[BC2_ENERGY]->clear();
        }
        stateDevice->setUsedBy(stateUsed);
1335    actuatorDevice->setUsedBy(stateUsed);
        if (!stateUsed) {
          double value = actuatorDevice->getLastValueSet();
          actuatorDevice->set(value, DataPoint::SET_SKIP);
        }
1340
        // M5 defines if S5 is enabled
        // We cannot rely on M6 BLEN measurement
        ++it;
        stateDevice = (*it);
1345    ++actIt;
        actuatorDevice = (*actIt);
        if (it == _states->end()) {
          throw Exception("Cannot access S5, check feedback configuration");
        }
1350    if (actIt == _actuators->end()) {
          throw Exception("Cannot access A5, check feedback configuration");
        }
        stateUsed = false;
        if (_measurementStatus[M5] == VALID &&
1355            stateDevice->isUsed()) {
            stateUsed = true;
            hasState = true;
            ++measIt; // M6
            measurementDevice = (*measIt);
1360        if (_measurementStatus[M5] == VALID) {
              measurementDevice->setUsedBy(true);
            }
            else {
              measurementDevice->setUsedBy(false);
1365        }
        }
        // Get initial actuator value if state was just turned on
        // and feedback is in the ENABLE mode.
        if ((stateDevice->getUsedBy() == false &&
1370         stateDevice->isUsed() &&
             stateUsed == true &&
             currentMode == true) ||
            (_previousMode == false &&
             currentMode == true &&
1375         stateDevice->isUsed())) {
          Log::getInstance() << Log::flagFbckPv << Log::dpInfo
                             << "Longitudinal::selectStates() "
                             << "reading initial values for "
                             << actuatorDevice->getDeviceName().c_str()
1380                         << Log::dp;
          if (actuatorDevice->getInitialSetting() != 0) {
            throw Exception("Failed to get initial value for A5");
          }
          // Clear averages related to BC2 current
1385      _currentAverage[BC2_CURRENT]->clear();
          _blenBc2Average.clear();
        }
        stateDevice->setUsedBy(stateUsed);
        actuatorDevice->setUsedBy(stateUsed);
1390    if (!stateUsed) {
          double value = actuatorDevice->getLastValueSet();
          actuatorDevice->set(value, DataPoint::SET_SKIP);
```

```
    }

    // Now figure out the USEDBYLOOP PV for the last state (L3).
    // It is defined by the state of three BPMs: M7, M8 and M10
    // Beam must be going though at least one ofe the above BPMs,
    // but not through all. If low tmit is present on all,
    // L3 state is off.
    ++it;
    stateDevice = (*it);
    ++actIt;
    actuatorDevice = (*actIt);
    if (it == _states->end()) {
      throw Exception("Cannot access S6, check feedback configuration");
    }
    if (actIt == _actuators->end()) {
      throw Exception("Cannot access A6, check feedback configuration");
    }
    stateUsed = true;
    if (_measurementStatus[M7] == VALID &&
            _measurementStatus[M8] == VALID) {
        stateDevice->setUsedBy(false);
        throw Exception("Invalid measurements, TMIT high on M7 and M8, skipping cycle.");
        return -1;
    }

    // Figure out which measurements should be used
    ++measIt; // M7
    measurementDevice = (*measIt);
    if (_measurementStatus[M7] == VALID &&
        _measurementStatus[M10] == INVALID &&
        stateDevice->isUsed()) {
      measurementDevice->setUsedBy(true);
    }
    else {
      measurementDevice->setUsedBy(false);
    }

    ++measIt; // M8
    measurementDevice = (*measIt);
    if (_measurementStatus[M8] == VALID &&
        _measurementStatus[M7] == INVALID &&
        stateDevice->isUsed()) {
      measurementDevice->setUsedBy(true);
    }
    else {
      measurementDevice->setUsedBy(false);
    }

    ++measIt; // M9
    measurementDevice = (*measIt);
    if (_measurementStatus[M8] == VALID &&
        _measurementStatus[M9] == VALID &&
        _measurementStatus[M7] == INVALID &&
        stateDevice->isUsed()) {
      measurementDevice->setUsedBy(true);
    }
    else {
      measurementDevice->setUsedBy(false);
    }

    ++measIt; // M10
    measurementDevice = (*measIt);
    if (_measurementStatus[M8] == INVALID &&
        _measurementStatus[M10] == VALID &&
        _measurementStatus[M7] == INVALID &&
        stateDevice->isUsed()) {
      measurementDevice->setUsedBy(true);
    }
    else {
      measurementDevice->setUsedBy(false);
    }

    if (_measurementStatus[M7] == INVALID &&
```



```
            _measurementStatus[M8] == INVALID) {
        if (_measurementStatus[M10] == VALID) {
            stateUsed = true;
            _measurementStatus[M10] = M10ENERGY;
        } else {
            stateUsed = false;
        }
    }
    if (!stateDevice->isUsed()) {
        stateUsed = false;
    }
    if (stateUsed) {
        hasState = true;
    }
    // Get initial actuator value if state was just turned on
    // and feedback is in the ENABLE mode.
    if ((stateDevice->getUsedBy() == false &&
         stateDevice->isUsed() &&
         stateUsed == true &&
         currentMode == true) ||
        (_previousMode == false &&
         currentMode == true &&
         stateDevice->isUsed())) {
      Log::getInstance() << Log::flagFbckPv << Log::dpInfo
                         << "Longitudinal::selectStates() "
                         << "reading initial values for "
                         << actuatorDevice->getDeviceName().c_str()
                         << Log::dp;
      if (actuatorDevice->getInitialSetting() != 0) {
        throw Exception("Failed to get initial value for A6");
      }
      // Clear averages related to DL2
      _energyAverage[DL2_ENERGY]->clear();
    }
    stateDevice->setUsedBy(stateUsed);
    actuatorDevice->setUsedBy(stateUsed);
    if (!stateUsed) {
      double value = actuatorDevice->getLastValueSet();
      actuatorDevice->set(value, DataPoint::SET_SKIP);
    }

    _previousMode = currentMode;

    // Return error if there is no State Device in use!
    if (!hasState) {
      throw Exception("No valid measurements and selected states, skipping cycle.");
      return -1;
    }

    return 0;
}

/**
 * If a state USED PV and the actuator FBCK PV have different values
 * the state of the FBCK PV changes to the value given by the state USED PV.
 * This allows actuators to be released from the feedback control once the
 * operators disable the equivalent state calculation.
 *
 * @author L.Piccoli
 */
void Longitudinal::setActuatorFeedbackPvs() {
  // Set FBCK PV only if the feedback is ENABLED, i.e. mode == true
  if (_loopConfiguration->_mode == true) {
    int i = 0;
    StateSet::iterator sIt = _states->begin();
    for (ActuatorSet::iterator aIt = _actuators->begin();
            aIt != _actuators->end() &&
            sIt != _states->end() &&
            i < MAX_LONGITUDINAL_MEASUREMENTS;
            ++aIt, ++sIt, ++i) {
      // ATTENTION: Skip setting FBCK for A4/A5, BC2 Ampl/Phase.
      // This is done in the LongitudinalChirp class
      if (i != 3 && i != 4) {
```

```
            if (_actuatorFbckState[i] != (*sIt)->isUsed()) {
                _actuatorFbckState[i] = (*sIt)->isUsed();
                (*aIt)->setFeedbackPv(_actuatorFbckState[i]);
            }
          }
        }
    }
}

int Longitudinal::updateActuatorsDelta() {
  _updateActuatorsDetailStats.start();

  // Proportional
  _pDelta = prod(_usedFgMatrix, _latestMeasurements);

  // Integral
  _iDelta = prod(_usedFgMatrix, _latestMeasurementsSum);

  double pGain = _loopConfiguration->_pGainPv.getValue();
  double iGain = _loopConfiguration->_iGainPv.getValue();

  _delta = (_pDelta * pGain);
  _delta += (_iDelta * iGain);

  Log::getInstance() << Log::flagAlgo << Log::dpInfo
                     << "Longitudinal::updateActuatorsDelta() gains: "
                     << _usedFgMatrix(0,0) << ","
                     << _usedFgMatrix(1,1) << ","
                     << _usedFgMatrix(2,2) << ","
                     << _usedFgMatrix(3,3) << ","
                     << _usedFgMatrix(4,4) << ","
                     << _usedFgMatrix(5,5) << ","
                     << "pGain=" << pGain << ","
                     << "iGain=" << iGain
                     << Log::dp;

  _updateActuatorsDetailStats.end();

  return 0;
}

/**
 * Update the values for the actuators using latests energies and currents.
 *
 * Changes in actuator values are added to the last values successfully written
 * to the ActuatorDevices.
 *
 * @return 0 if all actuator values are within limits, -1 if at least
 * one of the values is outside the limits
 * @author L.Piccoli
 */
int Longitudinal::updateActuators() throw (Exception) {
  unsigned int limitFlag = 0;
  double values[10];


  updateActuatorsDelta();

    // Fill in the current time and pulseId
    epicsTimeStamp timestamp;
    epicsTimeGetCurrent(&timestamp);
#ifdef RTEMS
    evrTimePutPulseID(&timestamp, _loopConfiguration->_pulseIdPv.getValue());
#endif

    // Update actuators
    bool actuatorLimit = false;
    StateSet::iterator stateIt = _states->begin();
    int i = 0;

    for (ActuatorSet::iterator it = _actuators->begin();
            it != _actuators->end(); ++it, ++i, ++stateIt) {
        ActuatorDevice *actuator = (*it);
```



```
        double value = actuator->getLastValueSet();

        if ((*stateIt)->getUsedBy()) {
            // Change the actuator value only if MODE PV is ENABLE (true)
            if (_loopConfiguration->_mode == true) {
                value -= _delta[i];
            }
            values[i] = value;
            if (actuator->set(value, timestamp) != 0) {
              limitFlag |= (0x01 << i);
              actuatorLimit = true;
            }
        } else {
            actuator->set(value, DataPoint::SET_SKIP);
        }
    }

    if (actuatorLimit) {
      i = 0;
      _exceptionMessage = "Actuator(s) out of range:";
      for (ActuatorSet::iterator it = _actuators->begin();
           it != _actuators->end(); ++it, ++i) {
        if (limitFlag & (0x01 << i)) {
          std::ostringstream strstream;
          strstream << values[i];
          _exceptionMessage += ",";
          _exceptionMessage += (*it)->getDeviceName();
          _exceptionMessage += "(";
          _exceptionMessage += strstream.str();
          _exceptionMessage += ")";
        }
      }
      throw Exception(_exceptionMessage, Exception::ACTUATOR_LIMIT);
      return -1;
    }

    return 0;
}

/**
 * Check throught which BPMs the beam is passing through and set the
 * _dl1Path and _dl2Path accordingly.
 *
 * @author L.Piccoli
 */
void Longitudinal::setPaths() {
    // DL1 Path
    if (_measurementStatus[M1] == VALID) {
        if (_dl1Path != M1) {
            _energyAverage[DL1_ENERGY]->clear();
        }
        _dl1Path = M1;
    } else if (_measurementStatus[M2] == VALID) {
        if (_dl1Path != M2) {
            _energyAverage[DL1_ENERGY]->clear();
        }
        _dl1Path = M2;
    } else {
        _dl1Path = NO_MEASUREMENT;
        _energyAverage[DL1_ENERGY]->clear();
    }

    // DL2 Path
    if (_measurementStatus[M7] == VALID) {
        if (_dl2Path != M7) {
            _energyAverage[DL2_ENERGY]->clear();
        }
        _dl2Path = M7;
    } else if (_measurementStatus[M8] == VALID || _measurementStatus[M9] == VALI
D) {
        if (_dl2Path != M8) {
            _energyAverage[DL2_ENERGY]->clear();
        }
```

```
            _dl2Path = M8;
1685    } else {
            if (_measurementStatus[M10] == VALID) {
                _dl2Path = M10;
                _energyAverage[DL2_ENERGY]->clear();
            } else {
1690            _dl2Path = NO_MEASUREMENT;
                _energyAverage[DL2_ENERGY]->clear();
            }
        }
    }
1695
    /**
     * This method makes sure the actuator is selected before updating its
     * reference offset. If the state is not selected the actuator MUST NOT
     * send FCOM messages out.
1700 *
     * @author L.Piccoli
     */
    void Longitudinal::setActuatorsReference() {
      // Fill in the current time and pulseId
1705  epicsTimeStamp timestamp;
      epicsTimeGetCurrent(&timestamp);
    #ifdef RTEMS
      evrTimePutPulseID(&timestamp, _loopConfiguration->_pulseIdPv.getValue());
    #endif
1710
      bool setActuatorWithNoBeam = true;
      StateSet::iterator sIt = _states->begin();
      ActuatorSet::iterator aIt = _actuators->begin();
      for (; aIt != _actuators->end(); ++aIt, ++sIt) {
1715      if (sIt != _states->end()) {
            if ((*sIt)->isUsed() && _loopConfiguration->_mode == true) {
              (*aIt)->getInitialSettingFromReference(setActuatorWithNoBeam, timestam
    p);
            }
          }
1720  }
    }

    void Longitudinal::show() {
        if (_dl1Path == NO_MEASUREMENT) {
1725        std::cout << "  -> No beam" << std::endl;
        } else if (_dl1Path == M2) {
            std::cout << "  -> Beam dump (going through M2)" << std::endl;
        } else {
            std::cout << "  -> Beam through M1 ";
1730        if (_dl2Path == M7) {
                std::cout << "and M7 to A-Line";
            } else if (_dl2Path == M8 && _measurementStatus[M9] == INVALID) {
                std::cout << "and to BYKIK";
            } else if (_dl2Path == M8 && _measurementStatus[M9] == VALID) {
1735            std::cout << "and to LTU";
            } else if (_dl2Path == M10 && _measurementStatus[M10] == VALID) {
                std::cout << "through BPMVB3, not reaching LTU";
            } else {
                std::cout << "only";
1740        }
            std::cout << std::endl;
        }

        std::cout << "            / M3 \\          / M5 \\       /- M7 -------->> <ALINE>" << std
    ::endl;
1745    std::cout << "  - (S1) -+-- M1 (S2) -/    \\- M4 (S4) (S5) -/    \\- M6 (S6) -+-- M10 - M8 +- M9 -
    >> <LTU>" << std::endl;
        std::cout << "      \\- M2 --->> <BEAM DUMP>                                \\----->> <BYKIK>
    " << std::endl;

        std::cout << "  -> Measurement status: ";
        for (int i = 0; i < MAX_LONGITUDINAL_MEASUREMENTS; ++i) {
1750        std::cout << "M" << i + 1;
            if (_measurementStatus[i] == VALID) {
                std::cout << "+ ";
```



```
            } else {
                std::cout << "- ";
1755        }
        }
        std::cout << std::endl;

        std::cout << "  -> States used: ";
1760    for (StateSet::iterator it = _states->begin(); it != _states->end(); ++it) {
            if ((*it)->getUsedBy()) {
                std::cout << (*it)->getName() << " ";
            }
        }
1765    std::cout << std::endl;
    }

    /**
     * Show time statistics.
1770 *
     * @author L.Piccoli
     */
    void Longitudinal::showDebug() {
        _checkMeasurementsStats.show("  ");
1775    _selectStatesStats.show("  ");
        _configureMatrixStats.show("  ");
        _calculateEnergyCurrentStats.show("  ");
        _updateStatesStats.show("  ");
        _updateActuatorsStats.show("  ");
1780    _updateActuatorsDetailStats.show("  ");
    }
```