Automation and Improved control of Bundle pusher

Automation control

Philip Borndalen
Abstract

Automation and Improved control of Bundle pusher

Philip Borndalen

Newspapers are often bundled and packet on pallets to secure transport. Packing the bundles on pallets are often done with an automatic palletizer. For reliable operation, the bundle needed to have a correct position and orientation on the infeed conveyor. Wrong orientation can happen when the bundle is pushed from conveyor to the palletiser's conveyor. The main reason is wrong timing that leads to rotation of the bundle and to high pushing force that leads to tumbled bundles. The goal of this project was to make a prototype of a system that automatically adapts the pressure and signal when to push using two laser sensors. One sensor sends a digital signal if laser breaks, the other measures distance, e.g. bundle height. The software was developed on an Arduino Uno board, and two interface boards were made for voltage level conversions. The prototype system managed to change pressure for different bundle sizes and timed the pushing signal correctly.
Contents

1 Introduction ................................................................. 4
   1.1 Background .............................................................. 4
   1.2 Goal ............................................................................. 4
   1.3 Limitations ................................................................. 4

2 Theory .............................................................................. 5
   2.1 Microcontroller ............................................................ 5
   2.2 Finite state machines ..................................................... 5
   2.3 Circular buffer ............................................................ 6
   2.4 Photoelectric Sensors .................................................. 6

3 Method, procedure and software structure ......................... 7
   3.1 System design .............................................................. 7
      3.1.1 Overview .............................................................. 7
      3.1.2 Sensors ............................................................... 9
   3.2 Interface boards .......................................................... 9
      3.2.1 Power Supply ....................................................... 9
      3.2.2 Digital interface ................................................... 9
      3.2.3 Analog interface .................................................. 10
   3.3 Software ................................................................. 10
      3.3.1 Main loop ......................................................... 10
      3.3.2 Convey finite state machine .................................... 10
      3.3.3 Pressure finite state machine .................................. 12

4 Result ............................................................................. 13

5 Discussion ......................................................................... 14

6 References ......................................................................... 15

Appendices .......................................................................... 16
   Appendix A Figures .......................................................... 16
   Appendix B Code ............................................................... 17
Chapter 1

Introduction

1.1 Background

Newspapers and similar printed products are often delivered from printers in bundles loaded on pallets. Palleting is often done with automatic palletisers and many different brands are available. To ensure reliable operation, the bundles must arrive at the machine with the correct orientation and the correct position on the connecting conveyor. It works well, but if the bundle gets out of the position tolerances, the machine will stop and require manual correction. Some types of transport conveyors maintain the positioning of bundles while other types less so. In particular, conveyors, mainly intended for distribution directly to lorries, have difficulty maintaining the precision in the transition to palletisers conveyors. At Bold Printing Stockholm AB, one of the major newspaper printers in Sweden, has this type of configuration. Presently pusher force, pressure, is set manually and timing is done using a single optic sensor.

1.2 Goal

The goal is to design and produce a modification for the system to avoid disturbances in the palletising process caused by faulty positioned bundles on the infeed conveyor by added abilities of the pushing system:

1. Correctly measure the bundle length and position of the bundle to allow for a correct pushing (avoid rotating the bundle due to too late or early push)

2. Change the pressure to the pusher to deliver the right amount of force for different bundle sizes.

3. Check if the bundles are out of position for a good push. If out of position then let the bundle go to the end station.

1.3 Limitations

The project is also limited to hardware and software design. Evaluation of the system performance is out of scope. Project limited to measurements possible by analogue bundle height sensor and timing photocells. Electronics limited to standard MCU boards (Arduino) and interface electronics on experimental development boards.
Chapter 2

Theory

2.1 Microcontroller

Microcontroller (MCU) is a computer on a chip. With mean the chip has integrated a Central Processing Unit (CPU), memory and in/output (I/O) peripherals. Where peripherals are special hardware to do specific tasks.

Most MCU has interrupts which are a signal to the processor to handle something different. When an interrupt request receives, the CPU will save what it was doing, and den starts the associated processing to the interrupt. The processes are called an interrupt service routine (ISR). After the process, the CPU goes back and processing what it was doing before the interrupt. This make interrupts perfectly for tasks where timing is essential or where the need to wait on an event. As the CPU can do other things than checking that event has happened. The active checks are called polling [1].

Counters are one of the most common peripherals after I/O. A primary counter used to count events. If the event is the clock, the counter turns to a timer. To get a timer to count slower than system clock, often very fast, a clock prescaler is used. The prescaler works by setting the number of clock active edges for count as an event for the counter. Two particular common functions for counters is a compare unit and a capture unit. Compare unit has a value loaded in its register and when the register value is the same as the counter an interrupt can be triggered or used to turn on/off a pin. A capturer unit is reverse to the compare unit, when an event happens, like state change on a pin then the hardware will copy the counter’s value to capture units register. [2].

2.2 Finite state machines

Finite state machines, FSM, is a method to model a system behavior by states. Each state describes how the system reacts to different events. FSM gives an option to react differently on the same event depending on what happened before. As FSM has a memory of events that have happened, the need for embedding if/else can be reduced. Embedding if/else often lead to understandably code. The second reason for FSM is every state encapsulate the code and divide into smaller parts. Witch make easier to handle the code.

One way to implement FSM is to have functions that represent the states. A pointer saves the current state by pointing to the current state function. To change the current state, only make the pointer point to the new state function. Events can be passed as an argument to state function when the function is dispatched. To maximise effective use FSM use a queue for the events needed, as the event often is detected by interrupt [3].
2.3 Circular buffer

A data structure for implantation a first in/first out FIFO buffer is called circular buffer. A circular buffer is made of an array and two index pointers. The index pointer points at the first object and last object in the FIFO. When a new object inserted the pointer to the last object is incremented, this action is called a push. A pop is when an object is taken out of the buffer where the object on the front index returns and the front index is incremented. The advantages with circular buffer compared to an array are that there is no need to shift every object when popping. [4]

2.4 Photoelectric Sensors

A reflective photoelectric sensor works by sending laser beam and sense the reflected beam by a receiver. Often the reviver and transmitter is hosted in the same module and use the reflected light. Most basic application is to detect if something is blocking the light between the transmitter and reviver. If the beam is blocked the sensor gives a discrete signal. The distance can be measured with using triangulation. The reflected laser beam is focused on a plane of light sensors with a lens. The light sensor detects where the spot is on the plane this position correspond to the distance between object and transmitter[5].
Chapter 3

Method, procedure and software structure

The problem was divided in three parts and was solved in order. The first part was sending/timing the signal to the pusher. The second was to change the pneumatic pressure for the pusher. The third problem was to measure if the bundle is too much out of position to be pushed. The added system shall work on top of the existing control system. The existing system incorporate a tracking system based on bar codes to address the different bundles to different pushers. This means that for this specific pusher all bundles passing the sensors must be measured but the bundles that shall be used for calculations is known only when the existing system actually pushes that specific bundle.

The software was structured using objects to model physical objects. The interaction between object was modeled with FSM's. Modelling made it easier to build with small steps as the modelling of the problem gave the solution. Software development was done in C++ using Atmel Studio and GCC-AVR compiler.

To reduce the programming knowledge needed for this project the FSM is not used with event argument as described in section 2.2. Instead of an event queue, background loop always dispatches the state and then polls to determine if an event has happened. Some advantages with FSM is lost, but still encapsulate and make it easy to divide the problem. In short, it is a different method of the switch statement.

Software debugging using simulation with step and breakpoints to confirm the correct behaviour. Serial communication is used to print state info for live debugging. The code to print was Arduino’s serial firmware.

The Arduino Uno board was selected for its accessibility, documentation and have easy to use code for serial communication. After defining all the signals that would be needed by the software, the interface boards were designed. Analog filter design was stimulated in LTspice.

3.1 System design

3.1.1 Overview

The system seen in figure 3.1 consists of a controller board, Arduino Uno with an Atmega 328P MCU, a digital interface board and an analogue interface board. The interface boards are soldered on universal experimental boards. Two laser sensor, one reflective and one triangulation type. The physical position of the sensor can be seen in figure 3.3.
The basic principle is that when a bundle arrives on the conveyor, the two sensors measure the bundle’s height and length. The length, height and position is first used to determine whether the bundle is pushable or not. If the bundle is pushable the length and position is used to calculate the timing of the signal to the pusher controller (e.g., simulating the single optical sensor in the original system). If the pusher pushes the bundle (means that the measurements actually is done for a bundle intended for this pusher), then the length and height is used to calculate the pressure reference.
3.1.2 Sensors

The solution uses three sensors, an optical distance, a photo-reflective and a pressure sensor. The photo reflective is used to timestamp the front and the end of a bundle. The timestamps are used to calculate the time between the front and back of the bundle. This sensor is in this report called digital sensor.

The optical distance sensor is an ODSL 9/V6 from Leuze Electronic placed over the conveyor to measure the height of the bundle. If the height is in a giving range, a digital signal will be sent from the sensor. This signal is used in the same way as the digital sensor, to timestamp the back and front of the bundle. This sensor will be referred to as the analogue sensor in this report.

The optical sensors height and length values are also used to determine bundle size.

3.2 Interface boards

3.2.1 Power Supply

24 V supplied from the original control system is used to power the three boards. 9 V for the Arduino board is generated by a DC-DC converter on the analogue interface, 9V is used by the Arduino board line regulator to produce 5V to power MCU and the analogue part.

3.2.2 Digital interface

Digital interface to the MCU.
Conversion between 24V to 5V for digital inputs signaler and 5V to 24V output for the digital output. Input and output are referring to MCU. The conversion is done with optocoupler’s, with open collector and
pull-up resistors. On the five volt side, the pull-up resistor is part of the MCU, which is activated by software.

### 3.2.3 Analog interface

The analogue interface converts the 0-10 volt analogue signal from the pressure sensor to 0-5 volt with a voltage divider. The analogue signal from the analogue sensor has buffer op-amp, 2-pole Butterworth low pass filter with a cutoff frequency of 21 Hz.

### 3.3 Software

#### 3.3.1 Main loop

Overview of the main loop is seen in figure 3.4. Main start witch initialisation pins, interrupts, timer/counters, ADC, UART and all object. The first function call in the background loop is Convey’s fifoRoutine. Function fifoRoutine task is to check if the first bundle in convey FIFO should be poppet and see if pressure reference should change.

![Figure 3.4: main() flowchart](image)

For Convey::dispatch() and PressureValve::dispatch() dispatch the object’s FSM, see more below.

#### 3.3.2 Convey finite state machine

The conveys FSM is made of seven states, as seen in figure 3.5. The initial state is the wait state. The wait state polls to see if a time stamp is available and store the data in a bundle object. When received from the analogue or the digital sensor a transition to the wait digital respective wait analogue state happens.
In analogue and digital wait state the same is done as in the wait state, but only for the other data set. If the other data set is received then state transition to check bundle. When in figure 3.3.2 the \textit{no analog} and \textit{no digital} means no headstamp for analog respective digital has been received. If \textit{no analog} or \textit{no digital} then a state transition to the calculate time state.

Check bundle state, check if a bundle is out of position, e.g. different length received from analogue and digital sensor. Due to poorly shaped bundles, the used solution was limited to check if analogue time length is shorter than the digital time length. Example of poorly shaped bundles can be seen in figure 3.6. If a bundle is too far away from the correct position, then a state transition to the clean state is done, else a state transition to calculation time state.

Calculate time state, calculate the time when signal to pusher controller, by finding the time left for the bundle center to be at the correct spot. It done with the equation

\[
\text{signalTime} = \text{tailTime} + \text{TIME_CONSTANT} - \frac{\text{lengthTime}}{2}. \tag{3.1}
\]

to calculate the timing for the signal to the pusher controller. Where in the tailTime is the tail stamp of digital sensor if available, else the analog is used. If both sensors is used then the lengthTime the is the
average between analog and the digital. If only one is available then that one is used. The time constant is the time for a bundle reaching the sensor to the point of sending the signal and is produced by testing. Before state transit the time for the signal is loaded to a compare unit of a timer. This timer will latter call a interrupt that sends the signal. The next state will always be exit state.

The difference between exit and clean state is that exit put the bundle in the convey’s FIFO. Where clean state discard the bundle, which means no bundle is put in the FIFO. Both state always state transit to the wait state.

3.3.3 Pressure finite state machine

The pressure valve object’s FSM made of four states shown in figure 3.7. When wait state is dispatched a check if the difference between pressure and reference are not to large. If to large error then the required adjustment time is calculated. Since all pressure regulator adjustments need to be finished by a small increased pressure adjustment, an additional time is added to the adjustment time if its required to decrease the pressure. The current time is saved, corresponding pin is set and state transition to corresponding state, up or down.

Both up and down state do the same thing. Check if adjustment time has passed and then stops the movement.

![Pressure valve object’s finite state machine](image)

Figure 3.7: Pressure valve object’s finite state machine. Green arrow when event happens. Blue always happens.
Chapter 4

Result

The system is tested in two ways. The first test with two different types of bundles, tabloid and quarterfolded, coming on the conveyor. The bundle have both different length and height. The system was able to adapt the pushing timing and change the pressure for the different types.

The second was tested was in real production with tabloid bundles. Ideal bundles where pushed correctly while bundles with less good shape showed some offset timing but still as good or better than the original system. Detection of rotated bundles on feeding conveyor worked for ideally shaped bundles but as soon as bundle shape differs the detection fails.
Chapter 5

Discussion

The functionality of the system was verified on the actual production system. In order to evaluate if the improvement of the system gives the wanted benefits, production supervision, data gathering over a period of time and analysis is necessary. This was not part of the project. But as noted in result, there is a problem with the use of the time difference between two sensors to detected if a bundle is rotated. It happens that bundles are as in figure 3.6, which the palletiser can handle but the sensor system indicate as rotated and discard the bundle. Use of other sensors to detect rotated bundles need to be investigated for future use.

A improvement could be to control the pressure setting by evaluating the physical pusher behavior, e.g measure the time for the pusher from home position to end position, instead of using predetermined pressure settings for different bundle sizes. For then no table with pressure reference is need and types of bundles.

Some software parameters, settings and information from the system to operator would be beneficial but requires some kind of operator panel. This was out of scope for the project but is a possible extension for the future and is probably necessary for a parament solution.

Switching between the old and new system creates random interference for the MCU. If the interference is from ground, then probably hard to isolate. It’s noted that the system will work after a bundle has passed the sensors. This is not a ideal situation as only one bundle is needed to make the pallet unstable. A basic solution could be to reset the MCU when changing mode.
Chapter 6

References


Appendix A

Figurers

Figure A.1: Bundle just pushed.

Figure A.2: The board are in the box in left corner
Appendix B

Code

main.cpp

```cpp
1 #ifndef F_CPU
2 #define F_CPU 16000000U
3 #endif
4
5 #include "bsp.h"
6 #include "convey.h"
7
8 int main(void);
9
10 #define IS_SET(x, a) (x & a)
11 #define IS_CLEAR(x, a) ((x & a)==0)
12
13 enum DoneFlag {
14   sensorLaserHead = (1U<<0),
15   analogLaserHead = (1U<<1),
16 }

convey.h

```
sensorLaserDone = (1U<<2),
analogLaserDone = (1U<<3)
};

enum class FifoState {
    WAIT,
    NOT_PUSHED,
    PUSHED
};

class Convey {
    #define MAX_NUM_BUNDLES 4 // Max bundels in the fifo (min three)
    #define FIFO_WAIT_TIME 4000 // [ms] Wait time for a bundle in the fifo,

public:
    typedef void (*State)(Convey * c);
    FifoState fifoState;
    PressureValve pressureValve;
    Convey( void );
    Bundel& getFirst( void );
    Bundel& getLast( void );
    Bundel& popFront( void );
    void puchBack( void );
    uint8_t queueSize( void );
    bool isFifoEmpty( void );

    Flag checkFlag(DoneFlag x);
    Flag& getStatusFlag( void );

    void fifoRoutine( void );

    void static dispatch( Convey * me );
    void static initialTran( Convey * me );

private:
    // Variables
    Bundel m_convey[MAX_NUM_BUNDLES];
    uint8_t m_frontIndex;
    uint8_t m_backIndex;
    uint8_t m_statusFlag;
    State static stateP;

    // Function
    void heightSensorRoutine( void );
    void bundelSensorRoutine( void );
    void checkFifoFlags( void );
    bool isTimeOut( Bundel * bundelP );

    void static cleanBundel( Convey * me );
    void static tran( State newState );

    // States
    void static waitState(Convey * me );
    void static waitDigitalState(Convey * me );
    void static waitAnalogState(Convey * me );
    void static checkBundelState(Convey * me );
    void static calPuchTimeState(Convey * me );
    void static exitState(Convey * me );
void static cleanState(Convey * me);

#include "convey.h"

Convey::State Convey::stateP;

Convey::Convey(void) {
    m_frontIndex = 0;
    m_backIndex = 0;
    m_statusFlag = NOTHING;
}

// FIFO handling

Bundel& Convey::getFirst(void) {
    return m_convey[m_frontIndex];
}

Bundel& Convey::getLast(void) {
    return m_convey[m_backIndex];
}

Bundel& Convey::popFront(void) {
    Bundel& pointer = m_convey[m_frontIndex];
    if (m_frontIndex != m_backIndex) {
        ++m_frontIndex;
        if (m_frontIndex >= MAX_NUM_BUNDLES) {
            m_frontIndex = 0;
        }
    } else {
        Serial.println("Cv.POF");
    }
    return (pointer);
}

void Convey::pushBack(void) {
    uint8_t diff = abs(static_cast<int8_t>(m_frontIndex) - static_cast<int8_t>(m_backIndex));
    if (diff < MAX_NUM_BUNDLES - 1) {
        ++m_backIndex;
        if (m_backIndex >= MAX_NUM_BUNDLES) {
            m_backIndex = 0;
        }
    } else {
        Serial.println("Cv.BOF");
    }
}

uint8_t Convey::queueSize(void) {
    uint8_t i;
    if (m_frontIndex > m_backIndex) {
        i = MAX_NUM_BUNDLES - m_frontIndex + m_backIndex;
    } else {
        i = m_frontIndex - m_backIndex;
    }
}
else {
    i = m_backIndex - m_frontIndex;
}
return i;

//
bool Convey::isFifoEmpty(void) {
    if (m_frontIndex == m_backIndex) {
        return true;
    } else {
        return false;
    }
}

// Measurement Flag function

Flag Convey::checkFlag(DoneFlag x) {
    return (m_statusFlag & x);
}

//
Flag& Convey::getStatusFlag(void) {
    return m_statusFlag;
}

//
void Convey::heightSensorRoutine(void) {
    if (Bsp::compHasValue) {
        Bsp::compHasValue = false;
        if (Bsp::isInt0ToggRising()) {
            getLast().getAnalogTime().tailTime = Bsp::getCompTime();
            Bsp::setInt0Falling();
            m_statusFlag |= analogLaserDone;
        } else {
            getLast().getAnalogTime().headTime = Bsp::getCompTime();
            Bsp::setInt0Rising();
            m_statusFlag |= analogLaserHead;
        }
    }
}

//
void Convey::bundelSensorRoutine(void) {
    if (Bsp::capture1Flag()) {
        Bsp::clearCapture1Flag();
        if (Bsp::capture1TriggRise()) {
            getLast().getBundelTime().tailTime = Bsp::getCaptur1Time();
            Bsp::setBundelTriggFalling();
            m_statusFlag |= sensorLaserDone;
        } else {
            getLast().getBundelTime().headTime = Bsp::getCaptur1Time();
            Bsp::setBundelTriggRising();
            m_statusFlag |= sensorLaserHead;
        }
    }
}

//
void Convey::fifoRoutine(void) {
    if (isFifoEmpty() != 0) {

Bundel * bundelP = &getFirst();
checkFifoFlags();
switch (fifoState) {
case FifoState::WAIT :
    if (isTimeOut(bundelP)) {
        popFront();
        Serial.println("F.t");
    }
    break;
case FifoState::PUSHED :
    fifoState = FifoState::WAIT;
    pressureValve.lastPushed(bundelP);
    popFront();
    Serial.println("F.p");
    break;
case FifoState::NOT_PUSHED :
    fifoState = FifoState::WAIT;
    if (queueSize() > 1) {
        popFront();
        Serial.println("F.d");
    } 
    else {
        Serial.println("F.x");
    }
    break;
}
else {
    Bsp::clearNextBundelFlag();
}
}

void Convey::checkFifoFlags(void) {
    volatile bool pushed = Bsp::hasBundelPushed();
    volatile bool newBundel = Bsp::isNextBundelFlag();
    if (pushed) {
        Bsp::clearHasPushed();
        fifoState = FifoState::PUSHED;
    } 
    else if (newBundel) {
        Bsp::clearNextBundelFlag();
        fifoState = FifoState::NOT_PUSHED;
    }
}

//

bool Convey::isTimeOut(Bundel * bundelP) {
    uint16_t waitTime = Bsp::getSystemClock() - bundelP->getEnterTime();
    if (waitTime >= FIFO_WAIT_TIME) {
        return true;
    } 
    else {
        return false;
    }
}

//

void Convey::initialTran(Convey * me) {
    stateP = (&(waitState));
}

//

void Convey::dispatch(Convey * me) {
    (*stateP)(me);
void Convey::tran(State newState) {
    stateP = newState;
}

void Convey::waitState(Convey * me) {
    me->heightSensorRoutine();
    me->bundelSensorRoutine();
    uint8_t localFlag = me->getStatusFlag();
    Bundel * bundel = &me->getLast();
    if (IS_SET(localFlag, sensorLaserDone)) {
        bool check = bundel->checkSetDigital();
        if (check) {
            tran(&Convey::waitAnalogState);
        } else {
            me->getStatusFlag() &= ~(sensorLaserDone | sensorLaserHead);
        }
    } else if (IS_SET(localFlag, analogLaserDone)) {
        bool check = bundel->checkSetAnalog();
        if (check) {
            tran(&Convey::waitDigitalState);
        } else {
            me->getStatusFlag() &= ~(analogLaserDone | analogLaserHead);
        }
    }
}

void Convey::waitDigitalState(Convey * me) {
    me->bundelSensorRoutine();
    uint8_t localFlag = me->getStatusFlag();
    if (IS_SET(localFlag, sensorLaserDone)) {
        Bundel * bundelP = &(me->getLast());
        bool pass = bundelP->checkSetDigital();
        if (pass) {
            tran(&Convey::checkBundelState);
        } else {
            me->getStatusFlag() &= ~(sensorLaserDone | sensorLaserHead);
            tran(&Convey::calPuchTimeState);
        }
    } else if (IS_CLEAR(localFlag, sensorLaserHead)) {
        tran(&Convey::calPuchTimeState);
    }
}

void Convey::waitAnalogState(Convey * me) {
    me->heightSensorRoutine();
    uint8_t localFlag = me->getStatusFlag();
    if (IS_SET(localFlag, analogLaserDone)) {
        Bundel * bundel = &(me->getLast());
        bool pass = bundel->checkSetAnalog();
        if (pass) {
            tran(&Convey::checkBundelState);
        } else {
            tran(&Convey::checkBundelState);
        }
    } else {
me->getStatusFlag() &= ~(analogLaserDone | analogLaserHead);
tran(&Convey::calPuchTimeState);
}
else if (IS_CLEAR(localFlag, analogLaserHead)) {
    tran(&Convey::calPuchTimeState);
}

//-------------------------------
void Convey::checkBundelState(Convey * me) {
    Bundel * bundel = &me->getLast();
    Flag okFlag = bundel->getOkFlag();
    if (IS_SET(okFlag, BUNDEL_SENSOR_OK) && IS_SET(okFlag, HEIGHTSENSOR_OK)) {
        if (bundel->checkSlant()) {
            tran(&Convey::calPuchTimeState);
        } else {
            tran(&Convey::cleanState);
        }
    } else if (okFlag != NOTHING) {
        tran(&Convey::calPuchTimeState);
    } else {
        Serial.println("X.c:");
        tran(&Convey::cleanState);
    }
//-------------------------------
void Convey::calPuchTimeState(Convey * me) {
    Bundel * const bundel = &me->getLast();
    Flag okFlag = bundel->getOkFlag();
    uint16_t lenthTime;
    uint16_t tailTime;
    if (okFlag & BUNDEL_OK) {
        lenthTime = (bundel->getBundelTime().getDiff() +
            bundel->getAnalogTime().getDiff()) >> 1;
        tailTime = bundel->getBundelTime().tailTime;
    } else {
        if (okFlag & BUNDEL_SENSOR_OK) {
            lenthTime = bundel->getBundelTime().getDiff();
            tailTime = bundel->getBundelTime().tailTime;
        } else {
            lenthTime = bundel->getAnalogTime().getDiff();
            tailTime = bundel->getAnalogTime().tailTime;
        }
    }
    uint16_t pushTime = tailTime + TIMEKONSTANT - (lenthTime >> 1);
    bundel->writeTime(pushTime, lenthTime);
    bundel->loadBundel();
    ATOMIC_BLOCK(ATOMIC_FORCEON) {
        Bsp::newPTime = pushTime;
    }
    Serial.print("P:");
    Serial.println(pushTime);
    tran(&Convey::exitState);
void Convey::exitState(Convey *me) {
    me->pushBack();
    cleanBundle(me); // clear flags
    trans(&Convey::waitState);
}

void Convey::cleanState(Convey *me) {
    cleanBundle(me);
    trans(&Convey::waitState);
}

void Convey::cleanBundle(Convey *me) {
    Bundle *const bundle = &(me->getLast());
    bundle->clearOkFlag();
    me->getStatusFlag() = 0; // clear flags
}
private: // variables
    ms_t m_qStartT;
    TimeMesurment m_sensorBundel;
    TimeMesurment m_analogBundel;
    HeightMesurment m_height;
    uint16_t m_length;
    uint16_t m_signalPuscheTime;
    Flag m_okFlags;

public:
    Bundel ( void );

    void writeTime ( uint16_t pushTime , uint16_t lenthTime );
    bool checkBundel ( void );

    uint16_t getsigTime ( void );

    TimeMesurment& getAnalogTime ( void );
    TimeMesurment& getBundelTime ( void );

    HeightMesurment getHeight ( void );
    void writeHeight ( uint16_t newHeight );
    HeightMesurment getLength ( void );
    void writeLength ( uint16_t newLength );

    void setOkFlag ( OkFlag flag );
    Flag getOkFlag ( void );
    void clearOkFlag ( void );

    bool checkSetDigital ( void );
    bool checkSetAnalog ( void );
    bool checkSlant ( void );

    ms_t getEnterTime ( void );

    bool isWaitingInFIFO ( void );
    void clearWaitingInFIFO ( void );

    void loadBundel ( void );
};

#include "bundel.h"

Bundel::Bundel ( void ) {
    m_okFlags = NOTHING;
}

void Bundel::writeTime ( uint16_t pushTime , uint16_t lenthTime ) {
    m_signalPuscheTime = pushTime;
    m_length = lenthTime;
    m_qStartT = Bsp::getSystemClock ();
}

bool Bundel::checkBundel ( void ) {
    if ( m_sensorBundel. getDiff () - m_analogBundel. getDiff () < MAX_TIME_DIFF) {
        return true;
    }
    return false;
uint16_t Bundel::getSigTime(void) {
    return m_signalPuscheTime;
}

TimeMesurment& Bundel::getAnalogTime(void) {
    return m_analogBundel;
}

TimeMesurment& Bundel::getBundelTime(void) {
    return m_sensorBundel;
}

HeightMesurment Bundel::getHeight(void) {
    return m_height;
}

void Bundel::writeHeight(uint16_t newHeight) {
    m_height = newHeight;
}

HeightMesurment Bundel::getLength(void) {
    return m_length;
}

void Bundel::writeLength(uint16_t newLength) {
    m_height = newLength;
}

void Bundel::setOkFlag(OkFlag flag) {
    m_okFlags |= static_cast<uint8_t>(flag);
}

Flag Bundel::getOkFlag(void) {
    return m_okFlags;
}

void Bundel::clearOkFlag(void) {
    m_okFlags = NOTHING;
}

bool Bundel::checkSetDigital(void) {
    bool pass = m_sensorBundel.diffInRange(TIME_LOWER_LIMIT, TIME_HIGER_LIMIT);
    if (pass) {
        setOkFlag(BUNDEL_SENSOR_OK);
        uint16_t diff = m_sensorBundel.getDiff();
        Serial.print("D: ");
        Serial.println(diff);
    }
    return pass;
}
bool Bundel::checkSetAnalog(void) {
    bool pass = m_analogBundel.diffInInterval(TIME_LOWER_LIMIT, TIME_HIGER_LIMIT);
    if (pass) {
        uint16_t height = Bsp::getHeight();
        uint16_t diff = m_analogBundel.getDiff();
        setOkFlag(HEIGHT_SENSOR_OK);
        writeHeight(height);
        Serial.print("A:");
        Serial.println(diff);
        Serial.print("H:");
        Serial.println(height);
    } else { return pass; }
}

bool Bundel::checkSlant(void) {
    uint16_t diff;
    uint16_t bTime = m_sensorBundel.getDiff();
    uint16_t aTime = m_analogBundel.getDiff();
    if (bTime > aTime) {
        diff = bTime - aTime;
    } else { diff = 0; }
    if (diff < MAX_TIME_DIFF) {
        setOkFlag(BUNDEL_OK);
        Serial.println("X.y: ");
        Serial.println(diff);
        return true;
    } else { Serial.println("X.n:");
        Serial.println(diff);
        return false;
    }
}

Bsp::ms_t Bundel::getEnterTime(void) {
    return m_qStartT;
}

void Bundel::loadBundel(void) {
    Bsp::writeOutputCompare(m_signalPuscheTime, m_length);
    Bsp::enableOutputCompareInterupt();
}

uint16_t TimeMesurment::getDiff(void) {
    uint16_t t = tailTime - headTime;
    return t;
}

bool TimeMesurment::diffInInterval(uint16_t lowerLimit, uint16_t higherLimit) {
    uint16_t diff = getDiff();
    if (diff >= lowerLimit && diff <= higherLimit) {
        return true;
    } else { return false;
    }
}
```c
#include "bundel.h"

class PressureValve {
    #define PRESSURE_FACTOR 60
    #define MIN_MOVE_TIME 100 // [ms_t]
    #define PRESSURE_MARGEN 50
    #define PRESS_MAX_ZON 650 // zon where only an attempt to increase pressure.
    #define EXTRA_DOWN 300
    #define PRESSUR_START_DELAY 6000
    typedef uint16_t Pressure;
    typedef uint16_t Height;
    typedef uint16_t Length;
    typedef uint16_t TimeValue;
    typedef Bsp::ms_t ms_t;
    public:
        typedef void (*State)(PressureValve * me);
        public:
            PressureValve(void);
            void lastPushed(Bundel * bundelP);
            Bsp::ms_t timeToMove(Pressure error);
            uint16_t getLength(void);
            uint16_t getHeigt(void);
            void writeLength(uint16_t newLenth);
            void writeHeight(uint16_t newHeight);
            void static printPressure(void);
            void inline static fsmDispatch(PressureValve * me) {
                (*stateP)(me);
            }
    private:
        State static stateP;
        ms_t m_lastRunTime, m_moveTime, m_startTime, m_delayTime;
        uint16_t m_height;
        uint16_t m_length;
        bool m_haveTried;
        bool m_moveDown;
        #define NUM_BUNDEL_TYP 2
        enum class BundelTyp : uint8_t {
            TABLOID,
            ICA
        };
        BundelTyp bundelTyp;
        const static struct Bs {
            Pressure PRESSURE_REF[NUM_BUNDEL_TYP] = {500, 400};
            Height HEIGHT_HIGH[NUM_BUNDEL_TYP] = {550, 650};
            Height HEIGHT_LOW[NUM_BUNDEL_TYP] = {300, 450};
            Length LENGTH_HIGH[NUM_BUNDEL_TYP] = {7000, 5500};
        };
```
```cpp
#include "pressurevalve.h"

PressureValve::State PressureValve::stateP;
const PressureValve::Bs PressureValve::BundelStats;

// PressureValve constructor

// Set the state to delay and wait for 20 ms so pressure can be determine.
// m_startTime should be zero.
PressureValve::PressureValve(void) {
    m_delayTime = PRESSUR_START_DELAY;
    m_lastRunTime = 0;
    bundelTyp = BundelTyp::TABLOID;
    stateP = &PressureValve::stateDelay;
}

void PressureValve::lastPushed(Bundel * bundelP) {
    m_height = bundelP->getHeight();
    m_length = bundelP->getLength();
    if (isNewTyp(m_height, m_length)) {
        changeTyp(m_height, m_length);
        Serial.print("P.n:");
        Serial.println(static_cast<uint8_t>(bundelTyp));
    }
}

//
bool PressureValve::isNewTyp(uint16_t height, uint16_t length) {
    if (height >= BundelStats.HEIGHT_HIGH[static_cast<uint8_t>(bundelTyp)] ||
        height <= BundelStats.HEIGHT_LOW[static_cast<uint8_t>(bundelTyp)] ||
        length >= BundelStats.LENGTH_HIGH[static_cast<uint8_t>(bundelTyp)] ||
        length <= BundelStats.LENGTH_LOW[static_cast<uint8_t>(bundelTyp)]) {
        return true;
    } else {
        return false;
    }
}
```cpp
//
bool PressureValve::changeTyp(uint16_t height, uint16_t length) {
    uint8_t i = 0;
    do {
        if (height <= BundelStats.HEIGHT_HIGH[i] &&
            height >= BundelStats.HEIGHT_LOW[i]) {
            if (length <= BundelStats.LENGTH_HIGH[i] &&
                length >= BundelStats.LENGTH_LOW[i]) {
                bundelTyp = static_cast<BundelTyp>(i);
                return 0;
            }
        }
        ++i;
    } while (i < NUM_BUNDEL_TYP);
    bundelTyp = static_cast<BundelTyp>(0);
    return 1;
}

//
Bsp::ms_t PressureValve::timeToMove(Pressure error) {
    Bsp::ms_t moveTime = error * PRESSURE_FAKTOR;
    if (moveTime < MIN_MOVE_TIME) {
        moveTime = MIN_MOVE_TIME;
    }
    return moveTime;
}

//
void PressureValve::printPressure(void) {
    Serial.print("P:");
    Serial.println(Bsp::getPressure());
}

//
void PressureValve::stateWait(PressureValve *me) {
    Pressure error;
    Pressure pressure = Bsp::getPressure();
    Pressure pressureRef = me->BundelStats.PRESSURE_REF[static_cast<uint8_t>(me->bundelTyp)];
    if (pressure < pressureRef) {
        error = pressureRef - pressure;
    } else {
        error = pressure - pressureRef;
    }
    if (error >= PRESSURE_MARGEN) {
        bool inMaxRang = (pressure >= PRESS_MAX_ZON);
        me->m_moveTime = me->timeToMove(error);
        me->m_delayTime = 3 * me->m_moveTime;
        me->m_startTime = Bsp::getSystemClock();
        if (pressure < pressureRef) {
            if (inMaxRang) {
                if (me->m_haveTried) {
                    Serial.print("T.h:");
                }
                else {
                    me->m_haveTried = true;
                    Bsp::pressureUp();
                    Serial.print("T.u:");
                    tran(&PressureValve::stateUp);
                }
            } else {
                Bsp::pressureUp();
                Serial.print("T.u:");
                tran(&PressureValve::stateUp);
            }
        } else {
            Bsp::pressureUp();
            Serial.print("T.u:");
            tran(&PressureValve::stateUp);
        }
    }
```
else {
    Bsp::pressureDown();
    me->m_moveTime -= EXTRA_DOWN;
    me->m_delayTime += EXTRA_DOWN;
    Serial.print("T.d:");
    me->m_haveTried = false;
    tran(&PressureValve::stateDown);
}
Serial.println(me->m_moveTime);
}

void PressureValve::stateUp(PressureValve * me) {
    ms_t t = Bsp::getSystemClock();
    ms_t diff = t - me->m_startTime;
    if (diff >= me->m_moveTime) {
        Bsp::pressureStop();
        me->m_lastRunTime = t;
        Pressure pres = Bsp::getPressure();
        Serial.print("T.m:");
        Serial.println(pres);
    }
    tran(&PressureValve::stateDelay);
}

void PressureValve::stateDown(PressureValve * me) {
    ms_t t = Bsp::getSystemClock();
    ms_t diff = t - me->m_startTime;
    if (diff >= me->m_moveTime) {
        Bsp::pressureStop();
        Pressure pres = Bsp::getPressure();
        Serial.print("T.m:");
        Serial.println(pres);
        me->m_moveTime = EXTRA_DOWN;
        me->m_startTime = t;
        Bsp::pressureUp();
        Serial.print("T.u:");
        Serial.println(me->m_moveTime);
    }
    tran(&PressureValve::stateUp);
}

void PressureValve::stateDelay(PressureValve * me) {
    ms_t t = Bsp::getSystemClock();
    ms_t diff = t - me->m_lastRunTime;
    if (diff >= (me->m_delayTime)) {
        Serial.println("T.y");
        tran(&PressureValve::stateWait);
    }
}