Patterns for the Companion Website

Veli-Pekka Eloranta, Johannes Koskinen, Marko Leppänen & Ville Reijonen
Self-Tests *

Also known as Built-in Self-test (BIST)

"I am completely operational, and all my circuits are functioning perfectly." - HAL 9000

...there is a CONTROL SYSTEM, which has hardware components that may malfunction and wear out with use. Some of the functionalities provided by the control system are such that they are used every day to carry out the work of the machine. If this kind of functionality malfunctions the problem will be easy to notice, as these functionalities are active all the time. For example, if the boom of the harvester does not move anymore according to operator’s control, the operator will notice it immediately. However, some functionalities are meant to be used only in a specific situation. For example, safety functions may be activated as rarely as possible. Still, these functionalities are usually essential for safety of the system and they must be operational when needed.

Rarely used functionalities, such as safety functions, may silently fail as they are not actively used. It may be hard to detect these latent malfunctions.

Determinism is a key property of the operations provided by the control system. The same input should always lead to the same output as long as the state of the system remains the same. If the output differs from the expected one, the function is not working correctly. Usually malfunctions are caused by a broken device or a bug in the control application. However, in a complex system it is not always possible to distinguish which set of outputs corresponds to which inputs. In addition, often only a part of the possible input values are meaningful in the normal operating mode.

Usually the malfunction of the device can be detected only when the device is in use. However, some devices may be used for rarely needed functionalities. Therefore, the malfunction is not detected before the functionality is activated. This is unacceptable for the devices providing safety functionalities.

It should be possible to ensure that the machine is still fully functional after a part of it is updated or replaced with a spare part. For this, the machine should provide a user interface or other means for service personnel to get information about the health of the machine so that the service person does not have to know internal details about it. In this way, the service person can ensure that the service is successfully completed. Similarly, operators of the machine can trust the functionality of the machine they are operating.

Sometimes a malfunction is caused by several errors. To ensure correct operation of the machine, it should be possible to determine all errors causing the malfunction, not only the one which first showed some symptoms. Even if the most prominent error is corrected, it may be that the root cause error still resides in the system, breaking it again after a while.
Therefore:

For each device, design a test sequence consisting of inputs and their corresponding outputs. Run the test sequence periodically or once in a while, e.g., during every third system startup. If the test sequence fails, it triggers a failure notification.

A test sequence is typically a piece of software that tests the functionality of a device by sending various input signals to the device and verifying that the output of the device is correct. For this, each test includes indicators which specify if the test passed or failed. Usually these indicators are related to certain observation points where the outputs of the device are inspected. For example, an operation of an actuator arm in a machine could be tested so that the hydraulic valve moving it is opened. If the arm is functioning, it will reach a limit switch within a certain time limit. Therefore, if the switch is closed, the arm and the limit switch are functioning normally and the test is passed. Otherwise, the test is deemed to have failed, as either the valve is not functioning, the arm is stuck, or the limit switch is broken.

Because the test sequences are based on the stimuli for the device and the expected responses of the devices is known, there needs to be an observer to check if a test is passed. Usually this is a software- or hardware-based system on the device, which sends the input signals for the device or its application one by one and compares the output signals to predefined ones. If a test fails, the testing system sends a failure notification to the rest of the system (see for example NOTIFICATIONS pattern) and the system may enter SAFE STATE if needed. This enables automated self-tests, as the device can test itself without any human intervention (see MINIMIZE HUMAN INTERVENTION, (Hanmer, 2007)). However, the observer could be also an operator of the machine. For example, all the indicator lights in the HMI of the machine could be switched on for a short time so that the operator can see that they are operating correctly. This can be combined with the self-tests of the devices so that a light is switched off only after the self-test of the corresponding device is passed. In this way, the operator can ensure the functionality of the light and the device. Human observers should be used mainly with HUMAN-MACHINE INTERFACE related tests, where it is otherwise hard to detect malfunctions. An overview architecture of SELF-TESTS is illustrated in Figure 1.

![Figure 1: SELF-TESTS send the input signals for the device and compare the output value to the expected one.](image-url)
The test sequences should be such that they really give relevant information about the health of the device. Usually this means that the normal operation of the device is simulated somehow. For this, a device may require additional sensors and loop-back capabilities to test the device without using other devices. In the case of actuators, testing a device may require physically moving it. In such cases, these tests can be executed only when the device is not operating. In addition, some tests cannot be carried out automatically, as they could cause harm to the environment or humans. These kinds of tests should require authorization so that they are executed only in a safe environment. See ROLE-BASED UI for more information.

Usually, system startup is a proper time to test the basic functionality and safety of the system. Tests can even be executed by BOOTSTRAPPER. If OPERATING MODES pattern has been applied, a new test operating mode can be added to the system. Separate test modes enable the operator or service personnel to start SELF-TESTS from the user interface without powering off and restarting the machine. This mode is used to run SELF-TESTS when the device is not in normal operating mode. This approach is also useful when carrying out long and comprehensive self-tests on demand. Naturally, there should be some kind of reminder to the operator or to service personnel to run these self-tests periodically.

Some tests are such that they can be executed even if the device is in normal operating mode and there is some idle time for the tests. Such a test can be interrupted if the device is needed for its normal operation, or the test can be executed as a long-running housekeeping task. Usually these kinds of tests are related to testing software and electronic components of the system, such as CPU, memory chips, or logic ports of a unit. For example, the application memory area of the unit can be so huge that there is not enough idle time to test the whole memory area in normal operating mode. However, it can be tested by applying PARTIAL RESULTS pattern so that the whole memory area is divided into smaller fragments. For each fragment, test functionality calculates a checksum from that fragment. Then the checksum is compared to the expected checksum. In case of any difference, the memory tests fails and a failure notification is sent. Similarly, one can apply the CONCURRENT EXECUTION pattern and have a memory test as a low-priority task, which is scheduled to be executed only if no other tasks need processor time. For more information on efficient RAM memory testing, see for example (Marinescu, 1982).

Some safety-related standards may require using SELF-TESTS. An example of this kind of standard is ISO 26262, which is a functional safety standard for road vehicles (ISO, 2011). However, SELF-TESTS can only detect if the device is malfunctioning at the moment of the test. For detailed analysis that can be used while a machine is in use, one should consider applying the DIAGNOSTICS pattern. There is also a standard, IEEE 1194.1 (IEEE, 2013), that defines a Standard Test Access Port and Boundary-Scan Architecture (JTAG) for testing embedded systems. By using JTAG it is possible to test the hardware, usually integrated circuit chips, for certain faults. However, JTAG is mostly meant for manufacturing to check if the system conforms to its functional requirements after assembly. In addition to testing a device itself, communication channels connecting the devices and working environment should be tested as well. See the FORCED INPUT VALUE pattern for details about how to find the root cause of a malfunction in a chain of units. In addition, HEARTBEAT can be considered as a test for a communication channel.

✥ ✷ ✷
Rarely used functionality can be tested periodically by using SELF-TESTS. Tests give information on the health of the devices providing functionality. This is especially important with safety mechanisms which are used only in case of emergency but must not fail when needed.

It may be impossible to provide comprehensive self-tests for all functionality. For example, airbags cannot be reused, so testing them must be carried out without triggering the actual safety mechanism. In addition, it may be impossible to have tests for all the parts in normal operating mode. Thus, they may only be tested in specific test operating mode. Therefore, SELF-TESTS cannot be the only mechanism to ensure functionality of the safety mechanism; regular tests made by an inspector are also required.

SELF-TESTS will reduce service costs, as a malfunctioning part may be pinpointed with comprehensive tests. At the very least, it is possible to ensure that a device is operating after a service operation. In addition, the broken parts cannot cause further damage to the machine, as a malfunction can be detected during startup time. In this case, the machine enters SAFE STATE as soon as the malfunction is detected. On the other hand, self-tests may require additional sensors, wiring, or other devices that are unrelated to the device’s main functionality. This adds manufacturing costs.

Some physical parts may require regular movements; otherwise, they may get jammed. This kind of movement is easy to carry out while testing the part at the same time.

(identifier)

An unmanned combine harvester can work autonomously with GPS guidance for short periods of time, but it requires periodic remote connection for exchanging work plans and production information. If the connection cannot be established, the harvester will stop. For this reason, it is essential to know that the hardware required for the remote connection is fully functional when entering the field. When the harvester starts up, it carries out several SELF-TESTS to ensure that the machine is completely operational. One of the tests is related to REMOTE ACCESS. During startup procedures, the main node of the harvester tells the communication device to self-test itself. The device puts itself in the specific test mode, which configures the device to loop-back all the communication locally. This means that the transmitter is internally connected only to the receiver of the device. Thus, all messages sent by the transmitter are received by the device itself. In this way it is possible to test transmitter and receiver locally without any actual remote connection. To test the functionality, the device sends a predefined sequence of messages and tries to receive them. If a received message differs from the transmitted one, the test will fail. In that case, a failure notification is sent to the main node (see NOTIFICATIONS), which enters the machine in SAFE STATE. The failures are also logged by NOTIFICATION LOGGING so that service personnel can easily find out the reason for entering SAFE STATE.

**Forced Input Value**

...there is a CONTROL SYSTEM, which has controllers with inputs and outputs calculated from the inputs. Usually the inputs come from sensors or the outputs of the other units. Similarly, the outputs are used as inputs of the other units or by the actuators. Thus, the units can form a chain of units, where the units are connected via a communication channel and each unit is responsible for its own
part of the whole. Naturally, if the input of the unit is incorrect, the output of the unit will be incorrect—even if the unit itself is functioning normally. This is also known as the garbage in, garbage out (GIGO) principle. Because of this, malfunction in a single unit will affect the whole chain following it and ruin the functionality provided by the chain. Based on the output of the whole chain, it is impossible to deduce which part of the chain is malfunctioning, as any of the control units in the chain or even the communication channel connecting them can malfunction.

Functionality consists of a chain of units, their inputs, and their outputs. In a case of malfunction in a specific functionality, it is hard to determine where the malfunction is by only monitoring the outputs of the chain. The malfunction can be in any of the control units or in the communication channel.

In a complex system forming a chain of units, it is hard or even impossible to know which part of the chain of units is malfunctioning. The unit can be functioning normally, but it produces incorrect output just because its input is incorrect due to failure in some earlier unit in the chain—or even in a communication channel. In addition, intermediate outputs from the chain, i.e., output from a unit in the middle of the chain, may be impossible to monitor if the proper monitoring points, such as hardware pins or global signals, for the output do not exist.

Self-Tests are typically carried out by verifying that the correct output is produced based on the selected input. In this way, Self-Tests can be used to verify that the chain of units is operating correctly. However, if a self-test of the chain fails, additional tests may be required to indicate which particular part of the chain is malfunctioning.

For comprehensive testing of a system, a set of various input values is required. However, in a real-world scenario, the available set of input values is typically more limited than required for comprehensive testing. For example, the temperature of the environment is not easy to change to suit the test cases.

Some of the units may be COTS components manufactured by a third party. These units are usually figurative black boxes, i.e., defined as only a set of inputs and outputs. It may be impossible to test that kind of unit other than based on its inputs and produced outputs.

In addition to controllers, physical parts of communication means may wear and malfunction. Erosion and corrosion in pins or wires can cause distortion to signals or even stop the communication completely.

Therefore:

Create a mechanism that can be used to force the control unit’s input to a specific value. This forcing mechanism is added to each control unit but separated from the communication channel. This makes it possible to check whether the output of the control unit receiving the forced input corresponds to the expected output. If the output is not correct, the control unit is malfunctioning, otherwise the communication channel is broken.
Dedicate an input channel which is used to force a certain input value. Setting an input value using this channel bypasses the actual input pin of the unit, thus forcing the input to have a certain known value from the unit’s perspective. Usually, this is done by a service person, who uses a channel with a dedicated tool. If the unit uses several inputs for operation, the forced input can be configured to be any of the inputs the unit uses. For safety reasons, it is common to force only one input value at a time, as this prevents the user from controlling the machine by using the forced signals.

Forced input value can be used to determine whether a certain unit or the communication channel is malfunctioning. The input value of the examined unit is forced to a defined one. The forced input can be anything from a single predefined value to complex input sequences. These inputs produce certain responses as outputs which can be monitored with an oscilloscope or the same tool setting the input. If the output of the unit is the expected one, i.e., corresponding to the input value, the communication channel is broken. Otherwise, the unit is producing the wrong output from the input. Usually this is caused by a broken device or a bug in the control application. Sometimes, especially with a state-based system, it may be hard to know the correct output value, as it may depend on the current state of the unit. In that case, the input value can be set to one that the real device, such as a sensor, is providing for that input. That way, if the output changes from the one provided by the real device, the communication channel, the terminals, or the producer is broken. A FORCED INPUT VALUE test can be repeated for each unit under suspicion until the malfunctioning part is found.

If it is possible to monitor only final output of the chain, the testing process will be different. Instead of providing input for just one unit and monitoring the output of the unit, the whole chain is used for testing. The test begins by setting the forced input for the first unit on the chain and monitoring the output. The process proceeds in similar way towards the end of the chain, unit by unit. When the remaining part of the chain produces the correct output from the forced input, the previous unit or the communication channel for the input is malfunctioning. This kind of pinpointing will reveal the unit or the communication channel where the malfunction exists. However, without knowing the output of the malfunctioning unit, the exact malfunctioning part cannot be known. Still, the number of suspected units has been reduced. Figure 2 illustrates this. The input of the chain is on the left side of the figure, while the output of the chain is on the right side. If the whole chain produces an incorrect value, the input of the unit B is forced for testing the output of the chain. If the output is now correct, then unit A or the communication channel between node A and B is malfunctioning. If the output is still incorrect, then unit B may be malfunctioning and the test is repeated by forcing the input value for unit C. The tests are repeated until the correct output is achieved or all the units have been tested.

![Figure 2: Example of finding a malfunction from a chain of units](image)
If the **Operating Modes** pattern has been applied, it is easy to require a certain test mode before any input can be forced. In this mode, the outputs of the units will not be used for controlling the machine. Even though this increases safety of the machine, it may prevent finding the problems that may occur only in a normal operating mode. It is also possible to design the system so that a separate tool is not needed and the service person can force the input values from **Human-Machine Interface**. In this case, the **Role-Based UI** should also be used for authorization. If the **Variable Manager** pattern has been applied in the system, the **Forced Input Value** pattern can be applied to force certain variables reflecting the inputs to the specific values. **Sensor Bypass** can be implemented using **Forced Input Value** and thus increase the fault tolerance of the system.

✦ ✦ ✦

With forced input values it is possible to test whether the unit produces the correct output value. This kind of test can be carried out even if the unit cannot be otherwise tested. For example, COTS components seldom provide any means to test the functionality of the system other than setting known inputs and monitoring the corresponding outputs. In this way it is possible to find out the malfunctioning part of the chain of units. The chain can be tested and the malfunctioning part can be identified even if only the final output of the chain can be received.

The corresponding output value for a certain input value should be known beforehand. This may be difficult if the expected output changes according to the state of the system.

**Forced Input Value** requires a dedicated channel for the input value. Using an HMI instead of a separate tool may require a lot of additional wiring, which may be cumbersome.

As the unit can be tested in place, the maintenance of the system will be easier. In addition, there is no longer a need for a separate test to discover which part of the system is malfunctioning.

✦ ✦ ✦

In a forest harvester, a set of controllers is responsible for all the movements of the boom. The controllers cooperate in a chain such that the first reads inputs from the sensors and filters the input values; the second one calculates the position of the boom based on the control algorithm, sensor data, joysticks, and pedals; and the third one controls actuators that actually move the boom. Suppose the operator complains that moving the joystick full left does not move the boom like it used to. A service engineer is sent to the field and she tries to identify the problem. The units contain a communication channel so that the engineer can set the input value of any channel to the desired value. Therefore, for the second controller, the output from the joystick is replaced with a value representing full left. The engineer notices that the output value of the controller is not the expected one. This means that the joystick and its wiring are operating normally, but the second unit is malfunctioning, so the controller is replaced with the new one. The tests are rerun and now the test shows that the machine is operating correctly.
Error Counter *

...there is a CONTROL SYSTEM with sensors and actuators. Because of disturbances in the work environment where the machine is used, sensor readings might be momentarily out of limits. In some cases, the sensor reading might not be received at all for a moment. However, detection of a transient fault such as a single missing reading does not necessarily mean that there is a persistent malfunction or fault in the system. Often a detection of fault will take the system to SAFE STATE and thus disable the whole machine. Because the availability and productivity of the machine needs to be maximized, it is not acceptable for a transient fault to put the system in SAFE STATE. The system is built to be robust and it can tolerate faults to some extent momentarily, so the response to a fault does not need to be immediate.

Because of demanding conditions in the environment, transient faults can occur in the system. These faults should not cause the machine to enter Safe State, as they clear themselves in a short while. Thus, substantial faults should be distinguished from transient faults that cause false alarms.

A transient fault should not prevent usage of a machine. If every minor transient fault, e.g., a sensor reading not received because of an oxidized connector, caused the system to stop or enter SAFE STATE, the availability of the machine would suffer. In addition, the machine operator would soon become frustrated because the machine is not functioning properly. Thus, small anomalies in system behavior should not immediately disable the machine. Conversely, all real malfunctions should be detected in order to ensure the machine’s safe operation. The system is designed so that the smallest malfunction will not cause further damages to the machine. In other words, system safety will not be compromised if the response to the malfunction is not immediate.

Some malfunctions or errors might be such that they do not reveal their true nature immediately. A malfunction looming in the system might first show some symptoms in the form of transient faults. However, these symptoms do not always reveal the real problem. Therefore, if remedying actions were taken based on the first symptoms, the real malfunction would not be detected. Thus, it might be advisable in some cases to monitor the situation for a while before taking action on the detected malfunction. This might enable finding the root cause of a fault. For example, suppose a battery is experiencing low voltage and the low voltage is causing the sensor to not provide the values. Taking an action on the first observable error, i.e., missing sensor reading, might not reveal the root cause, which is the low voltage of the battery. Furthermore, some malfunctions might be such that they cause momentary faults which are observable first, but after the momentary fault has passed, the true fault in the system is revealed. For example, static discharge might cause a threshold value of some measurement to be exceeded, but if the discharge has caused some other damages too, they might only be revealed later.

It is a waste of resources to process transient faults. If the fault will clear itself anyway, no remedying actions should be taken. For example, if a parity
check of data read from the memory in a satellite system fails, it might be advisable to try to read the data again instead of entering Safe State. The read operation might succeed on the second try because the first attempt might have been affected by cosmic radiation causing some bits to change.

Therefore:

Create an error counter whose threshold can be set to a certain value. The error counter is increased every time a fault is reported. Once the threshold is met, an error is triggered. The counter is decreased or reset after a certain amount of time from the last fault report has elapsed.

The basic idea is that an error counter monitors single or various events that indicate an error. Every time such an event occurs the counter is incremented. If the counter is used to monitor multiple events, each event might increase the counter value by a different amount. For example, message type A might increase the value by one whereas message type B might increase it by two because the message type B indicates a more serious situation. When the counter value reaches the threshold, a fault is triggered in the system. The most common strategy to decrease the counter value is to decrement it by one periodically.

Errors counted by the error counter can be almost anything: number of not received sensor values, seconds since the last message, etc. For each error counter a threshold value is specified. Threshold value indicates the limit causing a fault to be triggered. When the threshold is exceeded, the node enters SAFE STATE, using NOTIFICATIONS to indicate a fault or otherwise inform the rest of the system about the fault. Threshold values can be fixed values set at development time or they might be adjustable ones. Often PARAMETERS are used to set threshold values, as threshold values might change because of hardware wear and tear. The factory default threshold values are typically determined during the system testing phase when the machine is used in the field. The value is a trade-off between usability of the machine and fault detection response times. If the threshold is too small, usability suffers because the machine stops functioning when it encounters a transient fault. Conversely, if the threshold value is too big, the response time to an error situation suffers and might even compromise the safety of the machine.

The most common strategy is to decrease the counter value periodically at a certain interval. This is also described in LEAKY BUCKET COUNTER (Hanmer, 2007). In cases where this kind of error counter is used, typically it’s necessary to know that there are no more errors within a certain time unit than is allowed. For example, the sensor manufacturer states that if the sensor is producing more than 100 faulty results within an hour, the sensor is broken. The error rate of the sensor can be easily tracked by giving the error counter a suitable threshold value and decreasing it at the correct interval. However, there are other strategies for decreasing the counter, too. For example, the counter value can be set directly to zero after a certain period of time has elapsed since the last error. Furthermore, sometimes the counter value is decreased only when a successful event is detected. For example, the CAN bus uses error counters whose values are decreased upon each successfully sent message.

Figure 3 visualizes an error counter. In the figure, the error counter threshold is 10 errors. Currently the error counter value is three, as three errors have occurred recently. If altogether 10 errors occurred within a second, an error would be triggered. If new errors do not occur for a while, the counter value is decreased.
Sometimes it might be required to store the error counters in persistent memory. For example, if certain kinds of errors are so severe that they can occur only a limited number of times before the machine has to be maintained, then the counter value must be stored persistently. In these cases, one must design a mechanism to store the counter value. If there is no persistent memory on the node, the counter value needs to be transferred to a system PC for storing and restoring the value during \textsc{system} \textsc{start-up}.

If \textsc{notifications} are applied to the design of the system and the notification mechanism is designed so that the notifications have states, the \textsc{error counter} mechanism can be used to determine whether the fault is on or off. For example, when the threshold limit of the error counter is exceeded, the error notification (see \textsc{notifications}) is active and on. If the counter value decreases under the threshold, the fault is still on but marked as inactive. Naturally, both of these state changes generate their own notification. The \textsc{counters} pattern describes a mechanism to track different quantities in the system, e.g., usage hours and quantities tracked with \textsc{counters} are usually stored to persistent storage. The \textsc{riding over transients} pattern presented in (Hanmer, 2007) describes a mechanism to identify which faults are transients in advance. \textsc{riding over transients} might be implemented by applying the \textsc{error counter} pattern.

The system’s fault tolerance increases when error counters are used to distinguish actual faults from transient ones. Furthermore, the availability of the system is increased because the machine won’t stop immediately due to some small transient fault but can wait a little bit to determine whether there really is a fault in the system. Conversely, this might make the fault detection response times longer.

Some faults might cause symptoms that suggest a different problem in the system than the actual cause of the fault. When error counters are used, system resources are not wasted on processing these false faults, but the system waits for a while to detect the real cause of the fault. However, that waiting time might cause further damage to the system.

Threshold values for error counters might be sometimes hard to determine as they form a trade-off between fault tolerance and safety and response times of the system. Finding a correct threshold value might require extensive field testing and experimenting. However, the problem can be tackled to some extent by using \textsc{parameters} in adjusting the threshold values.

The operator should not be disrupted on each transient fault, as it would make the user experience worse. It is better to observe the situation for a while and only when there is certain substantial fault in the system should the operator be notified about the situation.

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{counter.png}
\caption{Visualization of an error counter. In practice, the counter is a normal integer variable whose value is monitored. If the threshold values are exceeded, a fault or other event can be triggered.}
\end{figure}
A forest harvester uses analog sensors to measure the grapple’s opening angle. The analog sensor cannot inform the CONTROL SYSTEM if the sensor is producing faulty values. The analog sensor does not necessarily have means to detect the faulty values. Sometimes the measurement can momentarily produce faulty values. Thus the ERROR COUNTER is used to detect whether the sensor is faulty or is just producing off-limit values momentarily. Listing 1 presents a code excerpt from a system. The sensor value is read using an analog-to-digital converter (ADC). Once the sensor value is converted to digital format, the system checks if the value is off-limits. If it is, the error counter is increased by one. If the value is within the limits, it is used in further calculations. Once the error counter exceeds the predefined threshold, the sensor is assumed to be malfunctioning. The system uses the SUPER LOOP (Pont, 2001) approach in the control and utilizes timer interrupts to decrease the error counter value once per second.

// Global error counter value
volatile int error_counter_value = 0;

// timer interrupt
ISR(TIMER_OVERFLOW) {
    if( error_counter_value > 0 )
    {
        // Decrease error counter value every 10 ms
        error_counter_value--;
    }
}

// Zero based ID of the ADC
double read_analog_sensor_value( int number ){

    // Read sensor value from ADC
    ADMUX = (1 << REFS0) | number;
    ADCSRA = (1 << ADEN) | (1 << ADSC);

    // Error Counter handling
    while (ADCSRA & (1 << ADSC));
    if( ADC < MIN_VALUE || ADC > MAX_VALUE ){
        disable_interrupts();
        error_counter_value++;
        enable_interrupts();
    }
    if( error_counter_value > THRESHOLD ){
        // Send error message or notification
    }...
}

int main(){
...
    while( true ){ // Super Loop
...
        value = read_analog_sensor_value( 0x0F);
        ...
    }
}

Listing 1: An excerpt from code reading analog sensor values and using an error counter to check that the value is within limits. Once the error counter threshold is exceeded an error message is sent.
Messaging Interface

Also known as Bus Abstraction

...there is a CONTROL SYSTEM whereby the nodes communicate in ONE TO MANY fashion using a HIGH-LEVEL PROTOCOL, such as CANopen. The selected protocol may become outdated and may need to be replaced with the new one, as the life cycle of a work machine is typically very long, in some cases over 20 years. The change in protocol might take place if the technology is not available on the market anymore or availability of spare parts decreases. In addition, the communication needs may change during the life cycle and the whole messaging infrastructure may need updating. For example, the number of measurements typically increases over time, requiring increased bus throughput; the security protocols may need an additional layer taking its share from the throughput. Furthermore, there might be new types of data, such as video images, which may require a totally different protocol. Changing the selected bus technology or communication protocol would necessitate changes to multiple nodes and multiple places in the code.

A high-level protocol may change over time, or the same control system software may be used in different communication setups. This should not require changes to the implementation of applications.

A HIGH-LEVEL PROTOCOL decouples applications from the details of low-level messaging and provides a common language for communication. Still, the selected communication protocol may have its own way to send and receive the messages, and the messaging concept may vary between various technologies. Therefore, even with the common protocol, the applications are dependent on the selected technology to some extent. Even when the communication channel technology is changed to another, it may be possible to use the old communication protocol with the new communication channel. However, this is not always an option because the new channel may not support the old protocol. Sometimes tunneling mechanisms are available which enable using the old protocol even though communication over the channel is carried out with some other protocol. If this kind of mechanism does not exist for the protocol or the new channel, the communication protocol must be changed with the communication channel.

The HIGH-LEVEL PROTOCOL forms a basis for all the communication. However, one protocol may not be optimal for all communication setups. As the applications are connected to the protocol interface, changing the protocol requires changes to the application code as well. In addition, this kind of coupling prevents reusing the same application code with the systems that have different communication means.

In order to enable a COMPONENT-BASED CONFIGURATION, the applications must be decoupled from the actual hardware. In practice, it should be possible to select the node where an application will be executed. In some cases, one node will execute more than one application (refer to CONCURRENT EXECUTION). Therefore, the communication between applications residing on the different
nodes should be same as that between applications executed in the same node. In other words, communication must be location transparent.

One protocol may not be enough for all communication needs. Thus, the systems can have various communication channels that use their own protocols. For example, for heavier communication needs, an Ethernet-based bus is used, while system status is communicated via a CAN bus. In this case, the applications must support all the protocols and their special characteristics. Therefore, instead of just one protocol stack (see the HIGH-LEVEL PROTOCOL pattern), the programmer must be familiar with multiple stacks. In addition, the application must know which communication channel and protocol should be used to reach the receiver. This is contrary to the idea of decoupling sender and receiver.

Therefore:

To make the application independent of the system bus technology and messaging protocol, construct a common application programming interface (API) to provide uniform messaging functionality. The API provides methods, such as for sending and receiving messages. Messages consist of data presented in the form of programming language structures.

Decouple the communication protocol and application with a new abstraction layer and design an API for it. The main idea of the API is to hide the actual communication implementation, bus technology, or protocol stack used for communication. With the common API, any of the aforementioned parts can be changed while the API stays the same. A change may require modifications to the implementation of the MESSAGING INTERFACE, but not to the application itself. The API will provide a uniform way of messaging, even if more than one protocol is used at the same time for communication.

The API should contain basic methods that enable the developer to use the communication channel in a uniform way. These methods typically include sending a message and receiving a message. For performance reasons, these methods should be asynchronous so that they will not interrupt execution of the application while sending or receiving the messages. Also, higher-level methods, such as sending periodic messages, can be included in the interface.

Implementation of the messaging interface translates these methods to those that are suitable for the actual messaging protocol and technology. Usually this means that MESSAGE QUEUE and HARDWARE ABSTRACTION LAYER are used to abstract the communication channel variations. For example, sending a message via Ethernet may require different preparations than sending the same message via CAN. In addition, HARDWARE ABSTRACTION LAYER abstracts nicely the differences between components from various vendors.

Two options are available regarding how to indicate the application which receives the message. The first option is to use point-to-point communication whereby the connection between two applications is first established. After the communication has been established, it can be referred to using a handle. All messages sent with the handle are targeted to the receiver application. A well-known example of this is communication with Internet sockets. The other way is to have a connectionless communication, whereby each message is individually addressed. The target application can be given as a parameter when sending a message or the message names can be unique so that the receiver is automatically known based on the name.

The communication should be decoupled from the protocol used. In practice, this requires that the messages depend only on the application, not on the
The selected communication protocol is used to exchange messages between the nodes. This enables the sender and receiver of the communication to work with different programming languages. The communication protocol is a binary representation of the message content. The sender can transform the data into a binary format, while the receiver interprets the binary data. This process is known as marshalling.

The communication protocol supports basic types such as integers, strings, and real numbers. In addition, more complex data structures can be sent, but either just a binary object containing the memory representation of the structure is transferred or the structure must be marshalled by the application before sending. Marshalling means that the memory representation of the structure is transformed to the data format suitable for transmission. It enables the sender and receiver application to be programmed with different programming languages, but it may require some extra communication capacity. At the receiving end, the application waits for the specific message and the API sets the variable given as a parameter when the message has been received.

The messaging interface must know which protocols and communication channels are used between the nodes, so a configuration file is used. The messages sent via messaging interface are either predefined in the configuration file or can be defined dynamically at run time—e.g., using common names for the messages and predefined types. If COMPONENT-BASED CONFIGURATION is used, the configuration file can be used to define all messages for the applications. In this way, the messaging interface is configured at the same time as the other system.

Listing 2 shows an example of a configuration file with two nodes, which are connected with the CANopen protocol (CAN, 2013). In addition, three messages to be used with the applications are shown in Listing 2.

```xml
<bus type="CAN">
  <protocol type="CANopen" messages="msg.cfg">
    <node id="1">
      <application name="BoomMovementApp"/>
    </node>
    <node id="2">
      <application name="MotorControl"/>
    </node>
  </protocol>
</bus>
```

Listing 2: An excerpt of the configuration files for MESSAGING INTERFACE. Two nodes using the CANopen protocol for communicating and three messages are defined.

The MESSAGING INTERFACE is similar to the HARDWARE ABSTRACTION LAYER (HAL) pattern. HAL abstracts hardware from the application while the MESSAGING INTERFACE abstracts communication. VARIABLE MANAGER is a communication abstraction pattern similar to MESSAGING INTERFACE.

MESSAGING INTERFACE provides an easy way to send messages between nodes, but VARIABLE MANAGER hides the whole messaging scheme and uses variables instead of messages. If VIRTUAL Runtime ENVIRONMENT is used, it usually contains some kind of abstraction layer, which abstracts the communication with real hardware. Still, the MESSAGING INTERFACE may be needed to abstract the
communication protocol. Listing 3 shows an example code fragment illustrating how the **MESSAGING INTERFACE** is used.

```c
/* Send a message to application in node 2 */
MessagingInterface mi;
mi.sendMessage("MotorControl", OIL_CTRL_NAME,
    "Oil Controller");

/* Receive a message */
char name[30];
while (mi.receiveMessage(OIL_CTRL_NAME,&name) != READ_OK);

/* Software update struct */
typedef struct {
  uint_16 version; // Software version
  uint_16 supported_hw; // Supported hardware id
  uint8_t code[1000]; // Application code
} ComplexType;

ComplexType application={...};
/* Update software in node 1 */
mi.sendMessage("BoomMovementApp",UPDATE_SOFTWARE,
    &application, sizeof ComplexType);
```

**Listing 3: Example code fragment illustrating sending messages with MESSAGING INTERFACE**

With **MESSAGING INTERFACE**, an application communicates with other nodes through the interface, which hides the actual communication channel and protocol used. In this way, the bus standard or protocol can be changed and the change will not affect the application code.

**MESSAGING INTERFACE** allows one to move the application from one node to the other without changing the application messaging code. This is essential for implementation of the COMPONENT-BASED CONFIGURATION pattern.

**MESSAGING INTERFACE** requires implementation for all supported environments and protocol stacks. This may be costly, may introduce additional points for errors, and may increase maintenance effort.

**MESSAGING INTERFACE** may require extensive configuration if multiple protocols and messaging channel technologies are used. One should consider a tool chain to configure the messaging interface automatically as part of the whole system.

Usually **MESSAGING INTERFACE** can provide only basic communication means, such as sending and receiving a message, as the interface is designed based on the least common denominator. Thus, this pattern is usually applied with other abstraction patterns, such as VARIABLE MANAGER. In this way, the control messages are sent via **MESSAGING INTERFACE** and system status messages are read from **VARIABLE MANAGER**.

Error situations, such as a broken communication channel, may not be noticed if the underlying communication protocol does not support such error detection. As a workaround, **MESSAGING INTERFACE** can contain its own way to
detect communication errors. For example, HEARTBEAT may be used to detect if the receiver application is still running.

A landfill compactor machine uses a CAN bus with CANopen for communication. When developing the control system, the real hardware is not always available or it is not feasible to use it. For this reason, the whole compactor’s hardware can be simulated in one PC. The communication is easy to change from actual CAN hardware to a shared memory-based system, as the control system uses MESSAGING INTERFACE to abstract actual communication. With the real hardware, sending the message is directed to the CAN bus driver. In the simulation environment, a memory block is received for the message and the receiving end just copies the information from the block. In this way, no changes to the application are needed when the system is changed between the simulated and real development environments. In addition, implementing the MESSAGING INTERFACE can provide tools for the developers, such as a messaging analyzer or simulated latencies.

Protocol Version Handshake

... there is a distributed CONTROL SYSTEM consisting of independent nodes communicating in ONE TO MANY fashion via a bus. As the system configuration may change over time, the nodes can be attached and detached independently from the bus. For communication, a HIGH-LEVEL PROTOCOL is defined and thus the nodes have a common messaging scheme. Over time, the messaging scheme changes to support new requirements and needs. For example, there might be a requirement to transfer more information for a new video camera feature, the size of a node identifier might be changed from 8-bit or 32-bit to support a larger number of nodes in the system, an new version of a sensor might return additional data which the older version of the sensor didn't measure, etc. However, it might not be possible to update a particular node to a newer protocol version if UPDATEABLE SOFTWARE pattern has not been applied or software isn’t available from the third-party vendor supporting the newer protocol. As a consequence, the nodes in the control system might be using different versions of the protocol, with the newer nodes using more recent software versions.

The nodes should use the latest version of the communication protocol, as it is probably the most efficient one. However, the system may also have nodes using older protocol versions and there should still be a way to communicate with them. Therefore, the most efficient version common to all the nodes should be determined.

Changes in a protocol which are not applied to both sides of communication may cause undefined behavior when the receiver has data in a format it is not able to correctly demarshall. There should not be any possibility for such an error.
Future needs are hard to predict and therefore it is difficult to design a “future-proof” protocol which would fulfill any later requirements. The protocol is clearer and efficient if current requirements are used as a base for the design.

Each node in the system should be able to communicate over a broadcast bus in **ONE TO MANY** fashion in such a way that all potential listeners can understand the content. When communicating peer to peer, both parties should understand what has been sent to them, and the protocol may vary depending on the communicating parties.

The machine manufacturer usually has no influence on third-party components and their software. Therefore, the protocol they use will stay the same unless the third-party manufacturer provides any updates or the component is replaced with a new one.

**Therefore:**

**Design a handshake sequence common to all protocol versions. During the handshake, all the nodes announce their highest protocol version at system startup. Once the nodes have announced their highest version, each node selects the highest common version for communication.**

During the handshake each node announces which protocol versions it supports. The **START-UP NEGOTIATOR** collects the protocol version information and announces the highest one that each node is able to use. This version number is then returned to each node to let them know what protocol version to use. An example of such a negotiation process is shown in Figure 4. The handshake process has to be defined so that the negotiator component and its handshake process does not change. Otherwise, there is no common ground from which the negotiation process could proceed.

![Figure 4: Handshake sequence in which nodes A, B, and C announce their versions to the negotiator, which in turn announces which version to use.](image)

Adding multiversion support makes the software more complicated. Translation from content into protocol, and back, should be encapsulated so that the protocol-specific code is not spread all around the code base. One should be careful to test that old protocol versions work correctly when the base of the code changes to take advantage of new features that in turn require a new
protocol version. When a node supports multiple protocol versions, difficulties will be encountered when the old protocol does not have a way to deliver required information or request activities that it requires from other nodes. In such a case the node cannot be used or the node has to be disabled.

To limit the number of combinations and reduce the testing to a manageable level, the machine manufacturer could announce that only a certain protocol version will be supported in the future. A manufacturer could also announce that some major versions of the protocol are supported for each machine generation, requiring the component manufacturers to provide different component versions for machines from different generations.

There are a few alternatives to supporting multiple software versions. A simple one would be to have UPDATEABLE SOFTWARE and update the old node to a software version which is compatible with the new node. More complicated would be a protocol translator from one version to another using a MESSAGE GATEWAY, but this might not work in cases where fast reaction times are required; and if one party is not able to request something that the other node can do, that functionality cannot be used.

A MESSAGING INTERFACE hides the actual protocol implementation, bus technology, etc., behind an API. This API would always be used the same way regardless of what is behind it.

Nodes which are based on different software versions are able to communicate with each other if a common protocol version has been found. Even new spare parts or new components can be added to the machine and it all should work without updating any software.

The protocol can evolve along with the rest of the system software and design in a piecemeal way. No time has to be wasted designing a “best possible” future-proof protocol. Still, supporting multiple protocol versions and/or translating content to different protocols adds overhead to the code and may complicate the codebase.

A lot of support code for older protocol versions might accumulate. Testing all supported protocol versions and their combinations becomes impossible when different hardware versions, software versions, and protocol versions accumulate over time.

The newer hardware might support some functions the older protocol doesn’t know and cannot therefore use. This might prevent using a node either partially or completely if the new hardware functions so differently that it cannot be operated using the older protocol. When both nodes are required, the work machine might be unusable.

The startup Negotiator must have a fixed protocol so that it is always able to function and determine the system state. This part of the system design cannot change in the future.

The original conveyor belt controller malfunctions on a ten-year-old stone crusher. A spare part controller is required but only new controllers are available. Maintenance is ordered on the site to solve the issue. A maintenance person replaces the old malfunctioning controller with a new controller. When the stone crusher is activated after service, the new controller announces the protocol versions it supports to the bus. Many of the versions are much newer than what the rest of the machine is capable of handling. The negotiator
component announces that the spare part controller should communicate with the same protocol version that the rest of the system has been using for the last 10 years, as it is the latest version that every node is able to use. When the machine is active, the maintenance person verifies that everything works as it should and releases the machine for use.

**Categorized Messages**

*Also known as Prioritized Messages*

...there is a CONTROL SYSTEM with a ONE TO MANY messaging scheme. There are several nodes on the bus, which need to communicate with each other, but the bus technology does not allow all nodes to send messages simultaneously. If two nodes try to send a message simultaneously, a collision will occur and time is lost retransmitting the messages. Also, it is not feasible to terminate writing a message to the bus once the sending has begun. Thus, other nodes on the bus should wait for a silent moment before sending anything. As all messages cannot be sent immediately to the bus, MESSAGE QUEUES are introduced to the system and messages wait in the queues for their turn to be sent to the bus. Therefore, delays in the system are unavoidable in messaging. However, this delay is not acceptable in the context of strict real-time requirements, as delays in the communication might result in delays in real-time behavior.

Because a message channel has limited throughput, not all messages will be delivered immediately. However, some messages relate to events, which may require immediate attention.

Messages relate to events occurring in the system, and these events require different response times. Thus, some messages are more urgent than others. Of course, all messages should be processed as soon as possible, but because of limited throughput of the messaging channel or the processing power of nodes, one should take into account the relative importance of the messages. For example, emergency messages should be processed immediately, whereas status messages may have more relaxed requirements and hence can wait a little bit longer until they are processed. Furthermore, while emergency situations should be taken care of as fast as possible, the system should also be able to return to normal operation as soon as the exceptional situation is over.

A message that needs immediate action may not discard other messages, as all messages should get through eventually. In other words, deleting or otherwise dropping out less important messages during the emergency situation is not an option.

A node in the system might need to react to an emergent situation by sending a message to the bus immediately. Unfortunately, the bus might be reserved when the message needs to be sent. Usually this means that a collision takes place on the bus and the higher priority node will get its message through. If the
node has a low priority this will cause problems as additional latency. To make things worse, repeated collisions cause latency to all delivery times if several nodes try to send a message simultaneously. Therefore, if multiple nodes have communication needs, multiple collisions will occur and the latency of communication will be further increased.

Therefore:

**Add a category to the messages according to their importance.**

*Importance can be based on type, sender or receiver, size, and so on.*

Separate MESSAGE QUEUES are implemented for each category.

To indicate the relative importance of a message, the message format should have some means to denote the message category. This message category is used to determine the order in which the messages are sent to the bus. The message precedence can be in the simplest form—only a field attached to the message indicating the urgency with a simple priority level. For example, one byte of the message can be reserved for the priority field, with byte value 0 meaning the most urgent messages, 1 slightly less urgent messages, and so on. Message importance is a relative concept so the designer must decide which message types are considered more important than the others. Usually error messages and other exceptional communication are given a higher priority. In the most simple case, there are only two levels of message priority: high and normal.

Usually it is not feasible to terminate writing a message to the bus once the sending has begun. Thus, other nodes on the bus should wait for a silent moment before sending anything. If congestion occurs, the nodes should stop sending low-priority messages in order to allow high-priority messages to get through. Congestion can be detected by listening to the bus or by an explicit congestion notification. To ensure that messages of high importance get through but no messages are lost, the messages have to be put into MESSAGE QUEUES. It is most efficient to have a separate message queue for each message importance level. In this way, no reordering of messages in a single queue is needed. In addition, no checking mechanism is needed for determining which message to process next. Because multiple queues are used, processing the next message is simply a matter of checking if the high-priority message queues have anything in them. If the highest-priority queue is empty, the second highest priority is checked, and so on.

When using multiple message queues, the node can send messages in the high-priority category even in the congestion situation, and the lower-priority messages wait in the queue for their turn. When the congestion is over, the lower-priority messages can be sent to the bus. See Figure 5 for an example, which shows two priorities, each of which has a dedicated MESSAGE QUEUE. If there are messages in the emergency message queue, it is emptied first. After the emergency queue is empty, the node can proceed with normal messages.

However, if there are plenty of high-importance messages, it may cause starving of low-importance messaging. Therefore, it could be sensible to add a time-to-live (TTL) field to the messages, so that messages that have waited for too long are not sent in vain. Of course, another mechanism can be used to solve this problem too.
The recipient must take into account that messages are no longer sent in the same sequence that they were created, so timestamping must be used if the order of messages needs to be preserved, such as for logging purposes. However, it might be sensible to also log the receiving order of the messages depending on the purpose of logging. In any case, the receiver should not assume anything from the order of messages when the CATEGORIZED MESSAGES pattern has been applied. The VECTOR CLOCK FOR MESSAGES pattern could be applied to determine the sending order of messages without using timestamps.

As there are many queues involved in the communication after applying this pattern, one should consider also adding an EARLY WARNING mechanism to ensure that queues do not overflow, or finding a suitable size for the queues during development and testing.

The CATEGORIZED MESSAGES pattern allows urgent messages to get through even in a congested situation. This improves the response times of the important messages and thus improves system safety, as the system can react to faults with shorter response time. However, determinism is decreased because the response time of a message depends on its priority and on how much higher-priority messaging is going on. This may lead to decreased testability because some messages may get attention only after a relatively long time and priorities make it difficult to predict timings for lower-level messages. In addition, the receiving order of the messages does not reflect their creation order anymore because the messages are no longer sent in same sequence that they were created.

Handling several message categories is a processor power- and memory-consuming task, as several message queues have to be kept in memory and checked every time a message is processed. However, within a message category, the order of messages is still retained.

It is quite easy to add new priority levels if a need for more fine-grained message categorization arises.

In some cases, priority inversion may occur. This means that the lower-priority message does not get through because of high-priority messaging, but the high-priority messages cannot run to completion because something is needed from the low-level messaging.
A power sweeper has controllers that communicate via a CAN bus. The messaging is organized such that in normal situations messages are handled with normal priority messages. However, if an emergency situation occurs—for example a controller malfunction is detected—then a high message priority is used to inform other nodes about the situation. These emergency messages have a higher priority than the normal messages so that other controllers can react to the situation quickly. In the case of a node malfunction, the reaction to the message is that every node goes into SAFE STATE and the system is basically stopped. The controllers will serve the high-priority messages first and enter SAFE STATE immediately. When the node malfunction has been remedied, e.g., by replacing the controller with a spare part, the messaging can continue normally after the system is rebooted. However, because the system requires reboot, all the lower-priority messages in the queues waiting for their turn are lost. Alternatively, these could have been processed while the system is in SAFE STATE. Naturally, in that case they should not initiate any movements.

**Message Channel Multiplexing** *

...there is a CONTROL SYSTEM using a **ONE TO MANY** messaging schema on a message bus. The message bus is inherently a shared messaging channel between multiple nodes communicating with each other. The communicating nodes have specified priorities—for example, using the id numbers of the nodes. The sharing of the communication channel causes problems because some of the participants may have difficulty getting their message through if other participants reserve the shared channel all the time. Either the messages collide with each other or, if the CATEGORIZED MESSAGES pattern has been used, the low-priority messages might not get through at all. This causes starving of those nodes which are sending lower-priority messages, as these nodes cannot communicate and proceed with their own functionality. Instead, they have to wait for the higher-priority messaging to cease so that the message bus is free. Similar starvation may take place if the node has a low priority dictated by the node’s id number.

To allow deterministic operation of a machine, it needs to be ensured that messages get delivered. A node may ‘babble’ to the bus and prevent other nodes from communicating.

Even low-priority messages should get through eventually. If the message is so irrelevant that it doesn’t matter whether it is received, then it is better not to send it at all. Thus, throughput for these messages should be guaranteed as well.

Determinism is often a key requirement in real-time control systems. Thus, messages which are related to the control operations should have guaranteed latencies. The latency should not vary significantly in order to ensure that the system’s real-time requirements are met in all circumstances.

If a node in a **ONE TO MANY** communication system starts babbling (see Babbling Idiot’s problem (Kopetz, 1997)) due to a malfunction, it can reserve
the whole bus if it can continuously send messages to the bus. If CATEGORIZED MESSAGES has been applied, the node can start to babble high-priority messages and prevent other nodes from communicating. On the other hand, even without CATEGORIZED MESSAGES problems may arise. If priority in communication is determined by node IDs, a malfunctioning node having high priority can prevent the lower priority nodes from communication at all.

If the selected bus technology allows collisions to occur freely, it is very likely that at least some of the messages will collide at some point. These kinds of collisions increase the delay of the messages sent by the lower-priority node.

Overall safety of the system can be compromised if bus latency is not deterministic or some nodes cannot communicate at all.

Therefore:

Separate the communication channel from the physical bus by creating virtual channels. Virtual channels can be multiplexed onto one physical channel by dividing the channel into time slots. A virtual channel can also be divided over several physical buses.

Provide a way for applications to use virtual channels for messaging. A virtual channel is an entity which acts like a physical message channel from the application’s point of view. However, a virtual channel is not bound to any physical medium. Virtual channels may be spread over several actual physical channels if one node has several physical mediums connected. The process of combining several signals or channels into one is called multiplexing. Especially in telecommunications, message multiplexing is used to describe a way to send messages from multiple sources over a single channel. However, in software engineering the term can have a slightly different meaning, using a single message to transmit data about several varied topics. In this pattern, the term is used as it is in the telecommunications domain.

One way to implement virtual channels is by adding a layer to the protocol stack, under the OSI model network layers (ISO, 1994). Using this approach, the virtual buses are built on the actual implementation of the added layer and can provide the same services to the network layer. See HIGH-LEVEL PROTOCOL for a similar approach. Virtual channels or subchannels (also known as tributaries) may also exist in a temporal dimension on the actual bus. Data that is meant to be sent in parallel is marshalled to the message channel. The message channel is divided into time slots and the data is put into these slots. This method is called time-division multiplexing. For example, a time unit, let’s say a second, will contain 30 message slots, each 33.3 milliseconds long. A slot may be wholly dedicated to a single node or to a full-duplex communication between two ends. In this case, virtual channels should have a configuration mechanism that allows adding and removing new links with simple PARAMETERS-based adjustment.

There may be different strategies for allocating a time slot if they are not statically dedicated in a SYSTEM START-UP via some configuration method. For example, a node may have a way to instantiate a call to a certain recipient by reserving a still free time slot. Then it must be ensured that the recipients listen to all slots to determine whether there is incoming data for them. All nodes have to honor the reservation of a slot until the repeating slot is freed with an end call message. If all the slots are reserved dynamically, there might be a situation where a node is denied a slot because all are reserved. Thus, the situation resembles the one without multiplexing.

As time-division multiplexing causes a notable portion of the actual bus capacity to be reserved for empty slots, there are ways to make the trade-off
between bandwidth and deterministic delivery time easier. One way is to provide slots for both real-time critical messages and more relaxed messaging. For example, periodic messages can have a certain time span reserved for them. The rest of the time before a new periodic cycle begins is fair game for all event-based messages. See Figure 6 for an example of this. In this scenario, three slots are reserved for periodic messages and the rest of the time is reserved for sporadic messages.

![Figure 6: Example of time division multiplexing. The bus is divided into time slots, each having a certain length. Certain time slots may be reserved for only certain types of messages (periodic and sporadic).](image)

There are further ways to multiplex communication over a single medium. Especially in wireless communication, it is also possible to divide the message channel into frequencies. This is called frequency-division multiplexing. However, this approach usually requires good hardware support and may be in conflict with standards and regulations. Thus, this is not typically the responsibility of a software developer and is mentioned for the sake of completeness only.

Using multiplexing may decrease the utilization rate of the message bus, as the maximum capacity must be high enough to allow all the virtual channels designed for the system even though they would not be currently in use. In some cases, the utilization rate may be as low as one percent. In addition, virtualization usually adds some overhead to the messages, which also contributes to the decreased efficiency.

However, the response times are more deterministic, as messages have a greater chance of getting through and no waiting is involved as long as the full capacity has not been used.

Fault tolerance against one node babbling to the bus and reserving it all the time is improved. Now the babbling of the node is limited to its own virtual channel, if no dynamic allocation of virtual channels is allowed.

The application developer does not have to consider physical messaging channels because the protocol stack takes care of the messaging details. All the developer sees is a virtual channel that connects the sender to the recipients. However, in some cases communication may be difficult to debug as the virtual channel may seem to work flawlessly, but lower-level errors in the physical communication medium may cause problems.

FlexRay (2005) is a common protocol used in machine control and automotive systems. It has time-division multiplexing, which is one instance of MESSAGE CHANNEL MULTIPLEXING. In FlexRay, the communication medium is accessed with a TDMA (time-division multiple access) scheme. The scheme is based on a fixed-length communication cycle, which is divided into four different segments: a static segment, a dynamic segment, and two additional...
segments called symbol window and network idle time. These communication cycles occur periodically whenever the system is running.

The static segment consists of several slots with fixed duration. The duration and the number of slots can be configured to the controllers with PARAMETERS. All controllers must share the same configuration in order to ensure proper operation of the FlexRay communication. These slots are assigned to the controllers so that each slot in a channel is exclusively reserved to one controller. This slot can be used by the controller to transmit message frames. This ownership relates to only one channel. On other channels, in the same slot either the same or another controller can transmit a frame. The recipients of the messages are not interested in the sender, but they are configured to receive frames in a specific slot. The static segment provides a way to send messages with deterministic communication latency.

Similarly, the dynamic segment also has a fixed overall duration. The segment is divided into minislots, which are shorter than the slots in the static segment. Their duration is so short that they cannot be used to send a whole message frame, but they provide a moment when a controller may start transmitting a frame. Each minislot is reserved for one controller in the network. When a controller has something to transmit, it waits for its own minislot and starts sending the frame. Other controllers see this and do not interfere with the transmission. Thus, the minislot is expanded into a similar slot as the static slot and can accommodate one message frame. After the frame is transmitted, the other controllers continue with counting the minislots and may transmit their own message frames. However, the expansion can reduce the amount of available minislots, so lower-priority controllers may not have time to send their own messages before the dynamic segment is over.

In addition to these communication segments, there are two optional segments for network management purposes. The symbol window is a special time slot with fixed duration where network management symbols can be sent. The network idle time is a time window when the communication channel is silent and the controllers can synchronize their clocks with a special algorithm. This time should be kept as short as possible, but long enough to have all clocks synchronized. The length of the network idle time slot is a parameter that is common to all the controllers.

The FlexRay communication may be configured to consist only of statically reserved slots by setting the duration of the static segment to fill the whole communication cycle. Conversely, if the static segment has only the minimum of two static slots, the messaging is purely dynamic. A mixed configuration is also an option, so the system designer has free hands in balancing the trade-off between bandwidth and determinism.

**Message Gateway**

*Also known as Message Channel Gateway and Converting Message Filter*

...there is a **CONTROL SYSTEM** with **ISOLATED FUNCTIONALITIES** communicating in **ONE TO MANY** fashion. However, as the functionalities may be different and have different requirements from each other, the optimal messaging solutions may be drastically different depending on the functionality. Thus, to optimize the collaboration within functionality, a suitable messaging technology should be selected between more tightly coupled nodes. The selected technology may differ from the best available technology for the other parts of
the system. Therefore, several **HIGH-LEVEL PROTOCOLS** might be needed to suit all these requirements.

**Parts of a machine may have different communication needs, so various messaging channels are required. The parts need to cooperate as a whole, so the messaging channels should be connected.**

In a subsystem the nodes must collaborate tightly, so they have to communicate a lot. In the whole system, the interconnection between nodes is looser and not so much communication is needed systemwide. Therefore, it is easier to design smaller subsystems which implement a smaller functionality together. In this way, the integration to the greater system can be carried out afterwards. Thus, the system as a whole is designed top-down and then isolated into subsystems. Conversely, the subsystems are designed bottom-up so that the local messaging is planned first and only afterwards systemwide.

Smaller parts are easier to master and they can be developed by a single person, making development efficient. Developing a tightly interconnected subsystem requires a lot of communication within the development team. Loosely coupled parts can be developed separately with less communication between developers.

The machine may consist of several subnetworks, each optimized for a certain need. However, the system must work as a whole, and the optimal solutions should have a way to exchange some of the communication with each other. The designer must balance the trade-off between the need for optimal throughput on the subsystem communications and the need for some messages to be sent to the other subsystems.

There are many commercial off-the-shelf products available that would suit machine control needs. However, COTS components usually support only a limited set of communication protocols and methods. If the whole machine were designed to support only a certain vendor-specific communication protocol, the threat of vendor lock-in grows.

In some cases, messaging is carried out over various media. For example, a sensor reading from the boom is sent to the controller with the ZigBee protocol (see e.g., [IEEE, 2011], [Gascon, 2008], or [Zigbee Alliance 2014]) over the air, the controller communicates with the CAN bus to the cabin PC, which in turn communicates over the Wi-Fi link to the control room.

All communication technologies have their own set of limitations, such as the length of a bus segment, the maximum number of nodes, and maximum throughput. Therefore, even if a single technology were suitable for the whole system, these limitations may require splitting the communication into smaller pieces that fit within the specification.

**Therefore:**

Add a message gateway component to the system which routes message traffic between message channels. If needed, the component can filter messages according to specific criteria defined in the system configuration.
In addition, the component translates messages from one protocol to another.

For each subsystem, design a node to be a router between each subsystem with a different message channel. This node is called a MESSAGE GATEWAY. The gateway component may be a dedicated node or a node which performs other functions as well. The gateway component is connected to the neighboring messaging channel, passing the messages from one channel to another and vice versa. This means that the gateway component has to know how to translate messages from one protocol to another.

In a ONE TO MANY approach, a sender node does not usually care who will read the message. Thus, the gateway usually replicates all messages to the other bus. In some cases, filtering may be needed due to security reasons, to simplify the design, or to decrease the messaging burden on the channel. The filtering may be carried out such that the gateway has a list of message types that should be passed on to the other network. The list can be either a white list, listing all messages to be passed on, or a black list, listing all messages that should be kept local only. Thus, it can apply the WHITELISTING FIREWALL pattern (Bonilla-Villarreal et al, 2013). The list may be based on the message type or another property, such as the sender. Therefore, the gateway has to be able to understand the protocol enough to determine whether a message is relevant on the other side. This processing takes some time, so a considerable latency can be involved when a message passes a gateway.

In some cases, the messages can have destination addresses. Then the message gateway can deduce from the address on which network the recipient resides. In this way, the gateway can discard those messages whose recipients reside on the same network from which the message was received, avoiding the unnecessary replication of messages to the other bus segments. In some cases, the nature of the messages has to be changed too, such as when one messaging channel sends sporadic messages and the other channel supports only synchronic messages. This adds considerable complexity to the implementation of the MESSAGE GATEWAY node, so it should be avoided if possible.

Usually gateways connect two networks together and messages are passed between them. In some cases, a separate backbone bus for the whole machine communication can be installed. In this case, each node connected to the backbone will act as a MESSAGE GATEWAY and pass all relevant messages to the backbone bus. However, a backbone bus needs additional wiring, making it prone to failure.

A variant of this is a local network, which has several message gateways. In this case, the network may act as an intermediary between several buses. Figure 7 illustrates two examples of MESSAGE GATEWAY usage. In example A, communication goes through the backbone bus and gateways connect the nodes to the backbone bus. In example B, there are several local buses which are connected through MESSAGE GATEWAYS to an intermediating network that has its own nodes. If the messages have dedicated recipients, some networks can be used just to route messages to their recipient. However, any messaging forms a coupling between senders and recipients. When the MESSAGE GATEWAY has to know the semantics of the messages it is passing, an additional coupling with the gateway is born. If a message has to be passed through nodes which would not otherwise be interested in the messages, an additional coupling is formed. Thus, it should be avoided.
Figure 7: A is an example of a backbone bus and B is an example of connecting several networks together.

In cases where the MESSAGE GATEWAY needs to change the messaging scheme, the gateway must repeat the received sporadic message periodically when it is passed to the bus where the periodic communication scheme is used. Similarly, when the gateway sends a message to the bus supporting only sporadic messages, the gateway must send a sporadic message only once when the value of a periodic message changes. This, however, might be hard to implement, as the gateway should know the contents of the message.

As a practical example, the CANopen Device Profile Truck Gateway DSP 413 (CiA, 2001) defines a gateway which makes CANopen objects available both on the truck and the trailer. A COMPONENT-BASED CONFIGURATION can use the MESSAGE GATEWAY pattern to connect several virtual buses together.

When MESSAGE GATEWAYS are used, several messaging technologies can be used on a single work machine. This makes it easier for a designer to optimize the technologies for local needs without sacrificing the connectivity of the whole, as MESSAGE GATEWAYS bridge these separate technologies.

If a gateway component is a dedicated node, it will add costs. If the gateway functionality is handled by a node with other responsibilities, it may not have time to do either its functionality or the message passing, if its processor power is low. In any case, message passing takes time and adds some latency to the messaging.

Bridging all messages to all connected buses adds load on both sides, but if a filtering or addressing scheme is used, the design and configuration of a message gateway becomes burdensome.

In a straddle carrier, the engine control is implemented with the J1939 protocol, as it is the industry de facto standard. However, the rest of the machine uses the CANopen protocol, so control messages from the cabin need to be translated into the J1939 protocol to control the engine. Conversely, error messages and other sensor information to be shown in the HUMAN-MACHINE INTERFACE’s instrument cluster needs to be translated to CANopen. Therefore, a
dedicated gateway COTS component, a Warwick PE-552 CANopen I/O unit with a J1939 gateway, is attached to the machine. The node is capable of communicating in both protocols and has two CAN connectors for connection to the base machine bus and the engine bus.

**Unique Confirmation**

...there is a Control System in which nodes communicate in One to Many fashion. Designers are implementing an in-house High-Level Protocol on top of those provided by the communication channel. The communication channel used is unreliable and can fail to deliver the message. Without an acknowledgement the sender cannot know if the message was successfully received. In addition, some messages may contain commands that initiate an action, and the sender needs to know if the action was started or not. In the protocol implementation, various messages need acknowledgement. In addition, sometimes multiple nodes may send an acknowledgement message, and in these cases the receiver might have a hard time identifying which nodes have sent an acknowledgement. For example, when the Operating Modes pattern has been applied, the system state change might be initiated by a single node. Other nodes may need to acknowledge this state change. If a single node is collecting these acknowledgements and ensuring that all the nodes are in the same state, a simple acknowledgement message might not be enough if the message is similar in all kinds of state changes.

The messages in the distributed system may be delivered out of sequence. If all the acknowledgement messages are identical, the receiver has no means to determine which request was acknowledged.

False confirmations to messages might have severe consequences and thus must be prevented. If an acknowledgement meant for one message is mistakenly interpreted to be meant for another message, an undesired action might be initiated unintentionally. To ensure safety of the machine such a situation must be avoided. In other words, it should be possible to know exactly for which message the confirmation was intended.

Some operations of the machine are such that they are not controlled by one message, but rather a sequence of messages to carry out the operation in smaller chunks. The requestor of such an action needs an acknowledgement that a part of the operation is initiated, so that the requester knows that it can now perform the next step in the sequence. If multiple operations are active in the system simultaneously, the requester needs to ensure from whom the acknowledgement came and for which request it was intended.

The order of received messages might change. If the CATEGORIZED MESSAGES pattern has been applied, the processing order of received messages might be different from the sending order. In addition, because of communication faults or CPU load, the acknowledgement message might be sent in a different order from how they were created. Therefore, the message receiver cannot assume anything about the order of received messages, so an
acknowledgement message can apply to any sent query, not only to the latest one.

The communication channel may cause latencies to the communication. In addition to the latency caused by the communication channel, the receiver of a request to initiate an action may be currently busy handling other requests. If the requester does not receive acknowledgement, it might re-request the action before deeming the receiver malfunctioning. In this case, the receiver might receive both requests, but should only initiate the action once. Thus, there should be a mechanism to distinguish between new requests and re-requests so that the operation is not initiated multiple times unnecessarily.

Therefore:

For each request message, assign a unique identifier, which is represented in a dedicated field. When the message is acknowledged, the response message contains the same unique identifier as the request. The requester keeps track of messages which are not yet acknowledged. In this way, the receiver of acknowledgement always knows for which request the response is for. In the case of a missing acknowledgement, the requester can resend the request.

When sending a request, the node assigns a universally unique identifier (UUID) to the request message. Basically, UUID is a 128-bit random number which is often presented as 32 hexadecimals digits. Therefore, a UUID is not guaranteed to be unique, but in practice it can be assumed to be. To further decrease the already unlikely probability of the same UUIDs, the sender node ID and timestamp of the message can be used in association with the UUID. In this case, the UUID will become even longer than 128 bits. There are multiple ready-made libraries that can be utilized to generate UUIDs. For example, an open-source project called OSSP UUID (OSSP, 2008) offers a ready-made library for C to generate UUIDs. There are also standards defining the properties of UUIDs and the algorithms to generate them. For example, ISO/IEC 11578:1996 (ISO, 1996) and the RFC 4122 (RFC4122, 2005) standard describe UUID and OSSP as well as many other implementations conforming to these standards.

In practice, the unique identifier does not necessarily have to be a UUID. A combination of node ID and timestamp, or node ID and a simple counter could be used. If a counter is used in association with node ID, it has to be ensured that the counter is not reset at SYSTEM START-UP. Furthermore, one must ensure that identifiers with the same timestamp are not created under any circumstances. As timestamps are often represented as 32-bit integers, this approach reduces identifier size and thus reduces the bus load.

The node sending the request keeps track of all its requests by storing the UUIDs of the messages. When the node receives a reply it removes the UUID from the list; or if the reply is not received within a certain time, the other node is thought to be malfunctioning and remedial actions can be taken. In this way, the node can see if it has received the acknowledgement message or some other reply to the request.

At the receiving end, when the node receives a request it processes it and creates a response or acknowledgement message to send back to the requester. The node then attaches the same UUID to the acknowledgement that the request message had. In other words, the receiver of the message does not generate a new UUID for the response. The receiver keeps recent UUIDs in memory. This way, the idempotence of the operation can be guaranteed, as if the request is
received soon again, it won’t initiate the action anymore. After a while UUIDs are cleared from the memory. The proper interval needs to be determined during testing of the system.

Figure 8 illustrates an example. The requester (node A) sends two messages (message 1 and 2) before the receiver (node B) has time to respond to either of them. If the acknowledgement messages are similar, then node A would not know for which message the response that node B sent is intended. However, as the request messages contain UUIDs and node B attaches that to the responses as well, node A can determine for which message the response is intended.

If the message is sent as a broadcast message to multiple recipients, the node has to know the number of recipients or this mechanism cannot be used. Usually this is not a problem, as it is adequate for one of the recipients to send the message acknowledgement. However, when using broadcast messages, it cannot be known which recipients have acknowledged the message—only the number of nodes can be known.

![Figure 8](image)

On lower levels of the OSI model (ISO, 1994) there might be a ready-made mechanism that implements the UNIQUE CONFIRMATION pattern. However, these are often used to ensure only that the message is delivered. On the application level, it is often important to also know if the action was really executed, not only that the message was delivered. Thus, this pattern is often applied when implementing an in-house HIGH-LEVEL PROTOCOL.

Unique confirmation improves messaging reliability because the requester can resend the request using the same UUID if the requester doesn’t receive the response within a certain interval. However, one should LIMIT RETRIES (Hanmer, 2007) to stop the requesting node from compromising message bus throughput by sending too many requests. If the requesting node does not receive a response after a couple of retries, the responding node may malfunction and the requesting node can start remedial actions, e.g., enter SAFE STATE.

If the recipient of the request receives multiple requests with same UUID although it has responded to the request earlier, it indicates that the acknowledgements are not being delivered properly. This may indicate a problem in the communication channel, and the node can start proper remedial actions, e.g., use notifications to inform the other nodes about the situation or increase the ERROR COUNTER.
ASYNCHRONOUS COMPLETION TOKEN (Buschmann et al. 2007) describes a similar mechanism for asynchronous calls, i.e., remote procedure calls or function calls from other threads.

A unique confirmation mechanism ensures that the sending node always knows for which message in the asynchronous communication the response or acknowledgement is intended. This helps to keep the system state consistent throughout the nodes in the system. UNIQUE CONFIRMATION does not help to detect the source of an error: Is the node malfunctioning or is there a problem in the communication channel? The node can only resend the request if the response is not received within a certain period of time, and must start other remedial actions if the response is not still received.

If the system logs all the messages, the system becomes easier to analyze when UNIQUE CONFIRMATION is used, as it can be readily determined from the responses for which request a message was intended.

Adding a UUID to every query-based message increases the message size and consequently increases the bus load. Sometimes, especially when using lower bandwidth buses, this can become an issue, preventing usage of the unique confirmation mechanism.

If the requesting node crashes, the bookkeeping of UUIDs is lost. However, this is not a big problem, as the rest of the system has probably deemed the node malfunctioning and entered SAFE STATE already.

A forest harvester synchronizes work plan and production report information with the FLEET MANAGEMENT system. The communication uses an application-level protocol between the on-board FLEET MANAGEMENT and the FLEET MANAGEMENT server. This protocol is developed in-house. The communication channel used is not highly reliable and packets may get lost in transmission or they may be received in a different order from the one in which they were sent. The on-board fleet management application needs to know which packets have been synchronized successfully to the server and which have not. The communication could be carried out in a synchronous manner but it would be too slow, as both parties would need to wait for the other party to answer before sending another message. Thus, a UNIQUE CONFIRMATION mechanism is implemented in the application-level protocol. Each time the recipient (server or machine) receives a message, it is acknowledged to the sender. Because message order is not preserved in the transfer of the messages, unique confirmation is needed to indicate which messages were received, so that the sender can resend those that were not acknowledged. Each work order, for example, is sent in its own message. The fleet management application on the machine acknowledges each successfully received work order with a separate message that has the same UUID that the work order message had. Both the fleet management server and the application on-board the machine utilize the Boost UUID C++ library (Boost, 2006) to generate 128-bit unique identifiers.
Sensor Bypass *

...there is a distributed CONTROL SYSTEM offering highly automated functionalities. These automated functionalities exist to control operations made by the operator and increase the machine operator’s productivity. Implementation of automated functionalities relies heavily on the sensor information and usually requires adding additional sensors to the machine. Additional sensors increase the probability of a sensor breaking down and when a sensor required by an automated functionality breaks down, the automated functionality cannot be used. If the sensor is broken down, LIMP HOME mode can be used to drive the machine safely to the nearest service. However, the machine cannot be fully operated anymore. Some sensors related to automation and enhancing the operator’s user experience provide relatively static information. If this kind of sensor breaks down, there could still be a chance to continue using the automated functionality at least to some extent.

Some sensors are used in a system design merely to provide small advantages in productivity or the operator experience. When this kind of sensor malfunctions, it can cause suboptimal operation or stop the whole machine.

Some sensors in the system are used to make it possible to automate functionalities in order to increase productivity or to enhance the user experience. The system could be operated without these sensors, but it would require additional control actions from the machine operator and decrease her productivity as the machine operator needs to compensate the missing sensor by monitoring the system more carefully. Thus, a malfunction of such sensor should not disable the whole machine. LIMP HOME could be used to drive the machine to service, but even LIMP HOME mode does not allow the operator to continue working.

If the machine breaks down in a forest or in another hard to reach place, it might take some time for a maintenance engineer to reach the location. In some cases, a maintenance engineer might have to travel from another country to get to the machine or the spare part might need to be ordered from the manufacturer. Thus, it may not be possible to replace the sensor immediately. However, down time of the machine is costly and thus should be minimized.

Therefore:

For each sensor of minor importance, implement a mechanism to replace the sensor’s output value with a substitute value. The value can be a default, a user-defined value, a simulated value, or the last known good value. In the event of a malfunction in the sensor, the substitute value can be used temporarily.
Define which of the sensors are such that their measurement value can be replaced with a substitute value, otherwise disable them. For example, if the outside air temperature sensor breaks down, the operator can acquire the approximate temperature from other sources and replace the sensor value with this reading. Create a user interface view for setting substitute values for sensors. Depending on the case, the user interface view can be accessed by the machine operator or only by authorized personnel such as a maintenance engineer. The ROLE-BASED UI pattern can be applied to restrict access to the user interface view.

Multiple options are available for setting the substitute value. One option is to use a default substitute value which is provided by the machine manufacturer. In this case, sensor bypassing means that the sensor value is substituted by this default or the last known good value. In this case, the user interface has a simple checkbox that enables or disables the sensor bypassing. Typically, this kind of user interface can also be accessed by the machine operator. A second option is to completely disable the sensor from the user interface view. When the sensor is disabled, the automated feature is not available at all, but the machine can still be manually operated.

In more advanced cases the machine operator or maintenance engineer can set the actual substitute value. In this case, the control system can suggest, for example, the last known good value as a replacement value. Of course, the machine operator might know a good guestimate of the value and can enter that. In the most advanced and complex case, the system can simulate the sensor value. Usually, this kind of simulation is based on the other sensor values and assumes normal behavior by the rest of the system. Therefore, it cannot be used if multiple system malfunctions are active. The simulation can be based on historical information collected from the machine or by models created during development of the system. While this kind of simulation approach might be useful, it might make other malfunctions or anomalies in system behavior harder to detect.

Figure 9 illustrates the basic principle of SENSOR BYPASS when using VARIABLE MANAGER. There are two signals in VARIABLE MANAGER: one indicating sensor value and another representing the status of the sensor. There is also a signal for substitute value, which is used in case of sensor malfunction. In the normal situation (1. in Figure 9), the sensor status has a value of 0XA0. Then the sensor breaks down and does not give a reading anymore. The node that reads sensor values detects the situation and sets the status signal to “fault” value (2. in Figure 9). This information also reaches other nodes on the system by using the normal VARIABLE MANAGER functionality. Then the machine operator decides to bypass the sensor and use a substitute value. She opens the user interface view meant for setting these substitute values. Now the sensor status is changed to “substituted,” but the actual sensor value is left untouched (3. in Figure 9). However, a third signal providing the substitute value is set to the value the operator has set in the UI view. Now, as the other nodes in the system read the sensor status signal and detect that the substitute value should be used, they can use the third signal instead of using the original signal containing the sensor value (4. in Figure 9). A separate signal for the substitute value is not required if a node can replace the value produced by a sensor, i.e., the sensor does not update VARIABLE MANAGER directly, but the node does the signal updating.
Sensor bypass can also be implemented without applying the VARIABLE MANAGER pattern. However, this likely requires implementing logic on each node which takes care of providing the substitute value. Of course, this approach might cause problems when the logic of using substitute values is changed because the changes are needed to all nodes. If the DATA STATUS pattern has been applied, it is quite natural to provide sensor status as meta-information about the variable containing the actual sensor reading.

Not all sensors are such that they can be bypassed. Some sensors are critical for the functional safety of the machine and thus should be never bypassed. For example, a sensor acting as a limit switch in a crane cannot be bypassed as it is directly safety related. In addition, when designing sensor bypassing, one needs to consider whether the prolonged use of bypassing might damage the machine. If so, it must be made clear to the operator that prolonged use of sensor bypassing might void the machine’s guarantee. Naturally, these sensors should not be visible in the user interface where the bypassing is enabled. However, sometimes even a mission-critical sensor can be bypassed by a maintenance engineer to allow limping home (see the LIMP HOME pattern for further details). Detecting malfunction of a sensor is not easy and in some cases might be extremely difficult.

The machine operator can still use the machine with automation functionalities enabled although some of the sensors required by the automation or convenience services is broken down. The automated functionality might not be completely accurate anymore, but it can still be used and the level of efficiency is not radically reduced. The system can continue using predefined automation sequences and algorithms although the precision might be lower.

The machine operator needs to be continuously informed that a sensor is being bypassed so she does not forget to order service for the machine. Sensor bypassing is meant to be a temporary measure, and using the machine for prolonged periods of time with a bypassed sensor should be discouraged.

Some system sensors are such that they are required to guarantee safe operation of the machine. Malfunction of such a sensor should put the system in SAFE STATE or LIMP HOME mode, and they cannot be bypassed even if this pattern is applied. In addition, local legislations might limit which sensors can
be bypassed. For example, legislation might require that the machine needs to be stopped if the cabin door is open. Therefore, if the cabin door sensor is broken down, it should not be possible to bypass it.

This pattern is about masking faults. Although the sensor can be bypassed by using the mechanism described in the pattern, the system still contains a fault that should be fixed, as it is probably working sub-optimally. Thus, the CONTROL SYSTEM should inform the machine operator clearly that a bypass mode is in use and service should be ordered as soon as possible.

A mining drill rig has automatic boom positioning functionality. The functionality automates the boom positioning process for the drill hole, so the machine operator does not need to do it manually. The automation algorithm requires installing three additional sensors on the boom, which introduces three new potential points of failure in the system. If some of the sensors malfunction, it should only disable the automation functionality as described in the LIMP HOME pattern. The SENSOR BYPASS pattern has also been applied and it is possible to replace the sensor values of the automated boom positioning functionality. If a sensor malfunctions, the operator can estimate, e.g., by using spirit level, the value that would be produced by the tilt sensor and enter a substitute value. Spirit level might be mounted to the machine by default for cases like these and to provide additional information for the operator when she is outside the cabin. This way, the automated boom positioning functionality could be used to roughly position the boom automatically and the operator only needs to manually fine-tune the position.

Counters

...there is a CONTROL SYSTEM with a VARIABLE MANAGER applied so data may be shared easily between the nodes in the system. There are different kinds of countable values which may increase, decrease, or reset depending on the machine usage. Such values and events could be, for example, total number of kilometers travelled, trip distance, motor usage hours, number of wheel rotations, and time elapsed since the machine was last stopped. These values might be interesting for the other nodes in the system and even to the machine operator. As the system is distributed, the consumer for the information might not be known, and in some cases there might be multiple producers for the same information.

The system typically has events whose number of occurrence is interesting. These occurrences are countable quantities that may need to be persistently stored. One event may affect several countable quantities, some of which are global in the system and some of which apply to only a limited set of the nodes. It is not always clear which node is responsible for storing the quantity.
There might be multiple nodes modifying the same quantity. Input should not be lost even if multiple nodes change the value simultaneously. Similarly, multiple consumers might be reading the value of the input, and they all should receive current information.

Only one node might be able to produce a quantity, as the sensor or actuator is physically located on the node. The only action which could be performed on it by others would be to read or reset it. Sometimes there might be a need to set limits for the quantity and react if a limit is reached, as it might signify some interesting occurrence.

Some of the countable quantities are persistent over the time, such as total usage hours, so powering off the control system should not erase such a value. Some of the quantities should reset when the machine is powered off. Quantity may also reset due to some other occurrence. For example, when a worn-out part is replaced the usage time should reset.

**Therefore:**

Create a service that provides counting functionality for different purposes. The service should offer different kind of counters—e.g., a non-resetting usage counter, a maintenance counter, and a resettable counter. The counters can count up or count down, even automatically.

Enhance the VARIABLE MANAGER with a counter service. The counter service can be implemented with simple asynchronous function calls to increase and decrease the counter value. The value modification can increase by one or by some other given amount. The value is stored in the VARIABLE MANAGER as any other value, but value modification is communicated to the other variable managers in other nodes as an operation of increase or decrease. Therefore value modification does not need to be an atomic operation, contrary to the normal way which VARIABLE MANAGER operates. Querying a value returns the value stored on the node’s local VARIABLE MANAGER component—the same way that it does with a vanilla VARIABLE MANAGER.

The counter value should be a linear quantity in case there may be multiple information producers. That’s because a change to a counter value could be made by multiple producers, resulting in simultaneous changes, as shown in Figure 10. A node which is responsible for storing and updating the value of a counter should be the node that has the software logic, sensor, or actuator producing the information.

![Figure 10: Example of simultaneous changes to a counter value](image)
Some of the counter values might need to be persistent. Consequently, when the programmer declares a counter, its life cycle has to be defined at the same time. The node producing the information would be a natural place to store the persistent data over restarts, but then changing the node might erase valuable data needed by the system. As all the values cannot be stored in all the system nodes, a backup location such as a high-end system PC should be used (see SEPARATE REAL TIME). In such a case the PC should announce the value of the counter over the bus when the system starts. For non-persistent data, an initial value should be defined, as not all counters start from zero and count upwards.

A counter might be resettable. Some of the resettable counters might be resettable by the machine operator and some might require operator-level usage rights (see ROLE-BASED UI). Along with the counter value, the time of resetting could be useful in many cases also. For example, a timing belt or sterile filter might have maximum amount of usage and maximum age. The time of reset would in this case provide a way to check if the maximum age has been reached. A similar idea can be explored even further for preventive maintenance in cases where a certain behavior is known to occur before failure, and diagnostic values may help to pinpoint such an occurrence.

When a certain threshold value has been exceeded and this threshold signifies some interesting occurrence, a notification should be triggered. By extending a Counter Service with NOTIFICATIONS we can alert interested parties about such an occurrence. To enable triggers, both the threshold and the action to take upon exceeding the threshold should be defined. The counter threshold is declared by the programmer or set by the machine operator from the UI (see PARAMETERS).

The trigger mechanism can be even further enhanced by implementing NOTIFICATION LEVELS so different triggers may be added to the same value based on their severity. By implementing EARLY WARNING an operator can be alerted before the threshold has been passed in cases where passing the threshold should not occur. A self-resetting counter can be implemented with an ERROR COUNTER for cases where an occasional fault can be tolerated, but many in a short time span should trigger an error.

In cases where the counter value change should be restricted only to certain nodes, please enhance VARIABLE MANAGER with VARIABLE GUARD to provide an access control mechanism.

Counters provide a simple method for handling values which should be shared over the system. The counter can contain useful information about the current situation, or the counter can be persistent over time depending on how long the information should be retained. A modular approach makes it easy to use the same mechanism in multiple places in the software. It also makes the software easier to analyze and understand.

There can be multiple producers making operations for the same counter, as the order of the operations does not change the result. When there are multiple producers in the system, counter values in some nodes might occasionally be a bit off before catching up. If an inconsistent operation such as a reset is required, a locking mechanisms will be required.

There is a possibility of counter value skew in a system if messages containing the counter value changes are lost. If there is only one producer for the information, value correctness can be easily verified by passing the actual value or a checksum. When multiple active producers are present, such an problem might be hard or impossible to notice.
It might not be simple to figure out where the value of a persistent counter should be stored when the system is powered off. Among the factors to consider are which of those multiple producers should store the persistent value and how should the value be backed up if the node storing the value is later replaced with a spare part.

The counter value for a cam belt increases every minute the tractor has the motor powered on. When the counter value exceeds the threshold limit in usage hours, a trigger is activated and a notification is sent to the cabin PC handling the operator user interface. To notify the machine operator, the cabin PC uses a blinking red warning icon on the interface and displays a notification indicating that the cam belt should be replaced because it has exceeded the normal usage hours. The machine operator notifies the service department and continues working. When a spare cam belt has been located, a service person replaces the old one with the spare. After replacing and testing the belt the service person resets the cam belt counter from the maintenance user interface.

**Start-Up Negotiation**

...there is a CONTROL SYSTEM which uses SYSTEM START-UP to ensure that all the nodes are present and functional before the system is started as a whole. This requires information about which nodes belong to the current system configuration and in which order they should be started. The configuration is fixed and defined, usually in a configuration file. However, system configuration may change during the lifetime of the machine—for example, because of installation of a new appliance. Thus, the system configuration file must be changed so that the new functionality provided by the appliance can be utilized. Usually this can be handled with a software update (refer to UPDATEABLE SOFTWARE), but updating the software is not a flexible approach to the problem and thus a bit cumbersome. To provide more flexibility, some of the devices may be defined as optional so that the system can start even if the optional nodes are not installed. However, functionality typically requires several nodes to be present, and the functionality can be used only if all the nodes are available.

A system’s configuration may change between start-ups. Some devices are mandatory to ensure safe operation. In addition, optional devices might provide additional functionality. The available functionality is hard to determine from the current set of devices.

System devices can be divided into two categories: mandatory and optional. Mandatory devices provide core functionality of the machine. Lacking any of these devices should prevent the system from starting. However, the system can
start if optional devices are missing. In that case, only some of the functionality may be missing. For example, if the air conditioning controller is not available, the machine can still be started and the operator can be notified about the missing device, but the rest of the machine functions normally.

Different functionalities may have safety mechanisms dedicated for them. If the functionality is not available, these additional safety mechanisms are usually not needed either. For example, if the harvester head is removed, there is no need for safety sensors to monitor feeding of the log towards the cabin. However, some of the sensors may be also used by other functionality. In these cases, it should be possible to ensure that removing a device does not jeopardize safety of the system.

Some control systems provide a way to install new appliances even though the CONTROL SYSTEM does not need to have special support for that particular appliance. However, the appliance may depend on some subsystems of the host control system. These dependencies must be handled before the appliance can be used. For example, a winch might require that the electric engine is available to provide required power. If so, the winching functionality can be enabled only if the engine is also available.

Therefore:

Create a mechanism in which all nodes announce themselves by sending a message to the bus after starting up. In the message, a node declares its existence and announces its capabilities. Design a central node, a negotiator, that gathers this information and builds a list of the nodes that are present. Using this list, the negotiator determines the available functionality. A node is ignored if it does not declare its presence within a specific time.

A device in the system provides some functionality for the machine. Capability refers to a set of certain functionalities. Some of the capabilities and functionalities are mandatory for basic system operation. If any of these capabilities are missing from the system, the system cannot be operated or even started. For example, devices related to safety functionality are required to be available before the machine can be operated safely. Other functionalities are optional; their absence does not affect the basic operation of the machine. For example, an appliance installed as an add-on provides some new capabilities for the machine, e.g., automated boom positioning based on GPS signals. These optional capabilities are not required when the system starts; but depending on the unavailable device, some functionality may be missing.

When the system is designed, its various capabilities are identified with an ID. The start-up negotiator determines which functionalities are currently available on the system, and based on this it decides which functionalities and capabilities can be provided for the operator. See an example of start-up negotiation in Figure 11. Nodes 1 and 3 are mandatory for basic boom functionality in the system, node 2 is required only for enhanced and semi-automated boom control functionality (boom control functionality marked with F1), and nodes 4 and 5 are needed for the boom head camera (functionality F2). If the negotiator finds that nodes 1, 2, 3, and 5 are present, it knows that 1) the system can be started, as all the mandatory nodes are present, 2) functionality F1
can be enabled, as node 2 is operational, and 3) functionality F2 cannot be activated, as required node 4 is missing. Even if node 2 were missing, the functionality of F1 could be activated at startup.

Figure 11: Example configuration with one mandatory and two optional functionalities and five nodes

In the simplest case, the capabilities of a certain node ID and requirements for the various functionalities are configured to the negotiator. Typically, the requirement configuration consists of the node IDs or capabilities of the nodes that are required to produce the functionality. Therefore, when the node starts to send HEARTBEAT signals to the communication channel, the negotiator knows that the node is present. From HEARTBEAT signals appearing during the start-up period, the negotiator knows which nodes are present in the system, and based on the requirement list it can determine which functionalities the system can offer, i.e., which requirements are fulfilled. This kind of predefined configuration is simple to implement, but all the supported devices must be configured to the negotiator component during the design phase. Therefore, it does not support devices that are published after the negotiator’s configuration file has been generated.

In a more dynamic implementation, no predefined configurations are used and the functionality is deduced based on information the nodes provide. During startup, the nodes send a message containing information about their capability IDs and any other capabilities that the node requires to provide functionality. Node IDs and a list of required (or mandatory) node IDs can also be used instead of capabilities. Based on this information, the negotiator can determine the current setup and available functionality.

Naturally, a proper timeout mechanism should be used so that the startup can continue even if some of the nodes do not send the information. Those nodes are simply ignored. If the ignored node is mandatory, this usually means that the system enters SAFE STATE and doesn’t start up. Figure 12 illustrates an example of message interchange between the nodes. After startup, the current operating mode (see OPERATING MODES) is set to negotiation mode. In this mode, the nodes send their ID and a list of the other devices they require for functioning correctly. Actions causing any movement of the machine are not allowed in this mode. After timeout, the negotiator sends the nodes whose requirements have been satisfied a message that changes the operating mode to normal operations. Nodes which do not have all requirements satisfied are powered off or other appropriate remedying action is taken.
In addition to checking the available capabilities, the startup negotiation should also check that software versions of the nodes are compatible with each other. For this, the node can attach the software’s version number to the capability ID message. If the software version is older than required, the negotiator can start the CENTRALIZED UPDATER to update the node, if possible. If no recent enough software version is available in the updater, the capability of the node cannot be used and the node is treated as if it were not present in the system at all.

If all possible configurations of the system are known, CONTROL SYSTEM OPTIONS can be used to enable and disable the functionalities. If an option is set for the system, it means that all the nodes related to that option must be present. For example, if a harvester has an option for a capsizing prevention system, the feature must be installed to the system with all required nodes; otherwise, the system may not start. Conversely, startup negotiation can be used to determine whether the required hardware is already installed but the option is not set.

If a TASK-BASED UI has been applied, the functionality determined by START-UP NEGOTIATION can be reflected by a HUMAN MACHINE INTERFACE. A missing part of functionality causes the user interface related to the missing functionality to be disabled or even hidden so that it doesn’t show to the machine operator.

With START-UP NEGOTIATION, the system configuration can change dynamically during the life cycle of the system. During startup, the negotiator ensures that all the nodes required by certain functionality are available before the functionality is enabled.

With the start-up negotiator it is possible to handle dependencies between optional subsystems. For example, an appliance may require some capabilities from the HUMAN-MACHINE INTERFACE so that the operator can control the implement.

System startup takes longer because the negotiator must wait for messages from the nodes. The messaging also means that all the nodes must be started before system functionality can be properly determined.

A generic capabilities vs. requirements system configuration is complex. The capabilities must be very detailed or some implicit device-type dependencies may appear. Usually it is much easier and safer to require that device X from
manufacturer A is needed than to try to define the required functionality in detail. In addition, testing a system becomes more complex or impossible to carry out exhaustively. In particular, standards for safety-critical systems may require that all the possible device combinations are analyzed and tested before the system can be approved, which might limit usage of this pattern.

A backhoe loader has a boom to which one can attach several implements—for example, a bucket or grapple. START-UP NEGOTIATION will inform the operator which implement is installed and the HUMAN-MACHINE INTERFACE will change depending on the implement attached. Depending on the implement, one or two joystick panels are required. Furthermore, depending on the model of the loader, the cabin may have a different number of joysticks and other controllers. In addition, the user can buy and attach additional control panels if needed. Because of this flexibility, the loader uses START-UP NEGOTIATION to ensure that the number and type of control panels correspond to those needed by the attached implement. When the system is started, the implements as well as the control panels will send their IDs to the negotiator. It uses a predefined configuration file to check if the number of joysticks is suitable for the implement installed. For example, a grapple requires two joysticks, so if the grapple is installed to the loader with only one joystick, a popup window is displayed, informing the operator that an additional joystick is needed, and the system will not start but instead enters SAFE STATE.

**Bumpless Update**

...there is a CONTROL SYSTEM where UPDATEABLE SOFTWARE makes it possible to change the system software after deployment. In addition, the CENTRALIZED UPDATER pattern might have been applied to allow a software update from the single update bundle. A software update usually requires that the system is stopped and taken offline for the update. However, some CONTROL SYSTEMs are such that they monitor and control a continuous process, e.g., a paper manufacturing process. Starting or stopping these kinds of processes is quite a long and demanding task. For example, breaks in the production might cause severe monetary losses, or a process might be required continuously without interruption, e.g., in a power grid control. However, from time to time the need to update the CONTROL SYSTEM software emerges.

The process that a control system is monitoring and controlling must not be interrupted. However, a control system software update causes a glitch in operation, as the update needs to be carried out when the controller is not in operating mode.

Maintenance breaks are costly in process control systems, so stopping the machine should be avoided. Usually these kinds of systems are taken offline once in a while for planned maintenance breaks, e.g., when bearings need to be
changed. During these breaks, the software could be updated as well. However, planned maintenance breaks are quite rare and in between these breaks it might be necessary to update the control system software, e.g., to patch security vulnerabilities. Sometimes the control system has such high availability requirements that it just cannot be taken offline to carry out the software update.

The controlled process might take long to time stop or start. Furthermore, as described in the DEVIL MAY CARE pattern, the process might have an unstable period during startup or shutdown. This unstable period often produces products which are low quality and cannot be sold to customers. For example, a paper machine might produce low-quality paper for a while when the process is started. Similarly, in nuclear plants the process takes time to stop. For these reasons, maintenance breaks should be avoided, as the starting up or shutting down of the process itself takes a lot of time and effort.

Even the 1+1 REDUNDANCY pattern cannot solve the problem of extraneous maintenance breaks during the software update if glitches at the process control are not allowed. Although the passive unit (see 1+1 REDUNDANCY for more information on division to units) could be updated while the active one takes care of the control and after update of the backup unit has been successfully carried out, the backup would be switched to be the active unit. However, this still causes a small glitch during the switch over and the state of the active unit might need to be synchronized to the passive unit. Sometimes synchronization is not possible because of large volatile data mass.

Nowadays, control systems are increasingly connected to the Internet to allow remote DIAGNOSTICS and monitoring (see the REMOTE ACCESS pattern). Connectivity has its price, as the software must be updated to patch it for security vulnerabilities. Otherwise, an attacker could gain control of the system and cause damage to it or even compromise the safety of the system (Jaatun et al., 2008).

Newer software versions could bring performance improvements and thus increase productivity. Alternatively, new software versions can also increase the quality of the service or product the machine produces by improving the system’s accuracy. However, even if the software update were small in size and quick to install, it would still mean a short break in the production process.

Software updates might be annoying for the machine operator if the update requires actions from the operator. Furthermore, the operator might not be an expert in software systems, so he or she might not feel comfortable with installing software updates. Thus, the update should be as invisible to the operator as possible.

Therefore:

Divide program code into functional blocks that have defined entry points. Create a system that updates the program on a unit block by block, so that the block currently executing is not updated. When control leaves the block, it can be updated on the fly.

The program code is divided during development into functional blocks. These blocks should be designed to be as independent as possible, but they still might have dependencies sometimes—for example, the blocks might require certain variables to exist or they produce certain variables. Therefore, when this is the case, the interfaces between the blocks must remain the same. Program code blocks can be updated independently of each other.

When a new software version is prepared, the update package is attached with information regarding which of the functional blocks needs to be updated
(see Figure 13). Sometimes a block is replaced with two blocks (blocks 2a and 2b replace block 2 in Figure 13.) Another way to determine which parts of the system need to be updated is to compare the current version to the update version block by block to see which parts are changed. During the software update, only the changed blocks are updated. To ensure that the execution does not reach the block being updated, the changed block is written to a new location in a memory. Once the update is written to the memory, the jump instruction in the code to the block is updated so that the program starts to use the new block from the new location instead of the old one. Naturally, the jump instruction is updated only when the execution is not running that particular instruction.

Sometimes there are dependencies between functional blocks, and blocks having dependencies are not compatible with old version. For example, if block 4 is updated, then block 5 needs to be updated as well because they have dependencies, and changes to outputs of block 4 are required (refer to Figure 13). Thus, both blocks need to be updated at once. In other words, this requires updating multiple blocks in a certain order to satisfy the dependencies and to ensure that the application is consistent during the update. Basically, this could mean that all the blocks are written first and the jump address to enter the first block is updated after all blocks are written and execution is somewhere else.

![Figure 13: Software is divided into blocks. Only the blocks which are changed are updated, while execution continues in the unchanged blocks](image)

Even when all dependencies, etc., are taken into account, BUMPLESS UPDATE is still not always possible. For example, if the CONTROL SYSTEM is such that it has a large code base for bootstrapping the controlled process and for fault handling but it has a relatively small control algorithm that is run in the loop, it is possible that the system cannot be updated without a bump. As the system is running the control loop most of the time, it cannot be updated. However, code for bootstrapping the process or fault handling parts might be updated without a bump. In these kinds of systems it is critical to retain the control loop unchanged over the updates. Furthermore, if the control loop calls other functions, it might be possible to update those in a bumpless manner. Parts of the system that cannot be updated for some reason should be extensively parameterizable, so that their behavior could be changed even without a software update.
In general, when using the BUMPLESS UPDATE pattern it helps if the code is designed to be stateless. If variable values are loaded into RAM memory, they might need to be changed, or new variables may need to be added, and so on. The VARIABLE MANAGER pattern could help to reduce the need for storing the variable values and state.

Sometimes new blocks are introduced in the update or old ones are removed. In this case, the updater must be able to handle the space requirements and determine to which block of the memory the application is written. If a new block is written to a new address in memory, the updater must handle writing the required jump commands so that the execution jumps to the right location. Alternatively, these can be handled in the updated code, if the update is specifically written for the control system at hand. In addition, sometimes some reserve space is left between the blocks in the first versions of the software to allow for the growth of the block size.

It might be necessary to create guard mechanisms to the program code in advance. These guards could help in updating the system—for example, when the update flag is on, the system does not try to jump to a specific block. Of course, this means that there are no availability requirements for that block.

Blocks sprinkled in various locations in code might cause fragmentation of the memory and after several bumptless updates there might not be enough space for blocks. In these cases, a clean installation is required to defragment memory space.

The BUMPLESS UPDATE mechanism is not suitable for safety-critical systems, as the software of those systems needs to be verified as a whole. During the update, the system might be in between versions and thus the current version is not safety certified. Applicability of the pattern for systems which are in constant interaction with a human operator, i.e., moving machines, is also limited. If the SEPARATE REAL-TIME pattern has been applied, the non-real-time part could be momentarily shut down for update, leaving only the machine control part to be updated in bumptless mode. If there are several different control algorithms and the INTERCHANGEABLE ALGORITHM pattern has been applied, algorithms not currently in use can be updated in the background. That approach resembles the pattern presented here.

✦ ✦ ✦

The system can be updated on the fly without taking it offline for update. In process automation systems, this provides flexibility in updating and costly maintenance breaks can be avoided.

The operator might not even notice the update at all. However, it is recommended that the operator be informed about the update in the HUMAN-MACHINE INTERFACE so that the new behavior or even new features do not come as a surprise. Furthermore, if the update process requires processing power to the extent that other machine operations might experience a lag, the operator should be asked before commencing the update.

Bumpless update makes the update process more complex. Implementation of the BUMPLESS UPDATE mechanism is demanding and thus prone to programming errors. Programming errors in turn might cause the bumpless update to fail and render the control system software in an inconsistent state and make it nonfunctional.

Some blocks, such as control loops, might not be able to be updated on the fly. Thus, these parts should be made extensively parameterizable using the PARAMETERS pattern so that their behavior can be changed without a software update.
As it might not be possible to update certain parts of the system, the interfaces must be stable. Interfaces cannot be changed during the update if there is a part using them that cannot be updated.

The interface of the program block must not change. If changes to the interface are required, blocks using that interface need to be updated as well.

A paper machine’s control system has a control loop whereby the process control is carried out. Listing 4 illustrates a simplified example of a control loop of this kind of system. The control system is divided into the functional blocks which are updated bumplessly. Initialization parts of the system are easy to update bumplessly while the control system is in normal operations. The parts which are part of the control loop are trickier to update. As shown in Listing 4, the control loop consists of different operations, such as reading sensor values, calculating the control values, and setting the outputs to the desired value. Thus, each of these operations is located in its own functional block, which can be updated while it is not being executed. Unfortunately, the algorithm is so long that it cannot be updated while the execution is outside of its block. Therefore, a separate flag can be set when the algorithm is being updated. When the flag is set, the algorithm is not used for calculating the values. This of course affects the control’s accuracy a little bit, as the algorithm is not executed for a couple of the control loop’s execution cycles. However, the system can be updated while it is running and the control is inaccurate only for a moment. Conversely, because CPU time is saved, it might help to meet the real-time requirements.

```c
while( true ) {
    retrieve_values(oldValue, oldValue2); // Block 1

    // Both sensor reading functions in the
    // same Block 2
    int tempReading = read_temp_sensor_1(); // Block 2
    double viscosityReading = read_viscosity_sensor_2(); // Block 2

    if( !update_algorithm_flag ){
        int outputValue = calculate_result(           
            oldValue, oldValue2, 
            tempReading, 
            viscosityReading); // Block 3
    }

    store_values(tempReading, viscosityReading); // Block 4

    set_output( PIN_09, outputValue); // Block 5

    // In the same block with set_output():
    set_analog_output( someValue ); // Block 5
}
```

Listing 4: Example control loop that could be updated block by block
Task-Based UI *  
*Also known as Task-Focused Interface*

...there is a CONTROL SYSTEM with a HUMAN-MACHINE INTERFACE. A machine has multiple capabilities such as moving, using a boom, etc. These capabilities require different control methods which need various gauges and controls on the user interface screen. Because the number of such capabilities might be large, the controls might not fit on a single user interface view at one time. As the operator can focus only on controlling a single capability at any given time, all functionalities are not needed simultaneously. Therefore, multiple user interface views could be used to solve the problem. However, navigating between multiple views could become cumbersome in the long run because the operator may need to switch between different views many times during a shift. For example, when cutting trees with the forest harvester, the operator needs to control the boom and harvester head, and from time to time needs to drive the machine to a new spot.

![Hammer and screwdriver](image)

There are several tasks with different requirements for the user interface. If a single user interface view is used to cover all these diverse requirements, it results in low usability. On the other hand, several UI views are confusing if the current UI view does not correspond to the task the operator is actually carrying out.

A machine might have many capabilities that the operator controls. The status information of these capabilities needs to be shown on the graphical user interface. However, these capabilities are often such that there is no need to use them all at the same time or the machine operator cannot safely operate multiple capabilities simultaneously. For example, a mining drill rig should not be driven while it is drilling, as it might damage the machine.

Controlling different capabilities may require utilizing the same physical controls because there is no room in the cabin to fit separate controls for each capability. This also limits the simultaneous usage of different capabilities.

Good usability is required from the machine. Functionalities of capabilities that are not needed at the moment should not make controlling the current task harder or confusing, but the relevant information should be easily found from the user interface.

If the OPERATING MODES pattern has been applied, the machine has different modes of operation that the control system keeps track of. For example, in the case of a mining drill rig, the machine might be in drilling mode or in driving mode, etc. These modes give the control system a context that can be used to determine which information and controls are important at the moment.

**Therefore:**

Design an operating mode for each task. Create a separate UI view for each task, and change the active view automatically when the operator changes the operating mode.
Each capability of the machine, e.g., using a boom, moving around, loading, remote control, etc., is assigned its own operating mode or context. Therefore, the machine becomes context aware. The machine operator can change the operating context by a physical switch or it can be automatically changed by the CONTROL SYSTEM. When the operating mode is changed, the graphical user interface also automatically switches to the corresponding UI view. For example, when the machine is in the boom control mode, the information and controls on the GUI also has controls only for controlling the boom. If the operator needs to access controls unrelated to the boom control, she can still access them by manually navigating in the user interface. In other words, other functionalities, such as adjusting parameters or accessing some other supporting functionality, should not be prevented unless the operating mode requires so. If the operations are forbidden in the current operating mode, these controls should not be accessible from the GUI either.

Figure 14 illustrates the principle of the TASK-BASED UI. The system has multiple different operating modes—e.g., drive mode, tree-felling mode, SAFE STATE and update mode, etc. The operator can change the operating mode between tree-felling mode and drive mode by turning a physical switch. When the mode is changed, the user interface changes too. In drive mode, the operator sees speed, engine RPM reading, and other driving-related information. In the tree-felling mode, there is information concerning the current log being sawed, and the operator can select for example the species of the tree being cut. When the operator plugs in a USB memory stick, the system automatically changes to update mode. In this case, the updating user interface is automatically displayed. The view contains controls for system update and the required user interface elements to visualize the update progress. As shown in Figure 14, update mode can be entered from both drive mode and tree-cutting mode. From the update mode, the system can return to the either one of the previously active modes. Furthermore, when the machine detects a fault it enters SAFE STATE automatically.

The primary goal of this approach is to scope the information shown on the user interface to only what is relevant to the operator’s current task. Different techniques can be used to determine which information is relevant to the task. One of the most popular ways to identify relevant information is to assign a degree-of-interest (DOI) ranking (see e.g., (Card & Nation, 2002)) for each
functionality and information element. This means that the more frequently the machine operator has to interact with an element (showing information or control element) while carrying out the task, the higher the DOI ranking the element has for the current task.

DOI rankings can be created, for example, by observing a machine operator carrying out tasks with the machine. Another option is to collect data on machine usage and analyze that to determine the DOI ranking. Of course, other methods could be used as well. Multiple machine operators could be observed to make the DOI ranking more reliable. When designing the user interface views for the tasks, these DOI rankings are used to select the elements that are shown on the task-based user interface view. Functionalities and elements having a low DOI ranking are filtered out from the user interface. Within a single task’s UI view, the DOI ranking can be used to select which elements are made larger and which should be easily accessed, and which ones need longer navigation sequences to access them.

Sometimes DOI rankings can be dynamically changed. In this case, the control system stores a log of elements that the operator interacts with and the user interface elements shown are determined based on this stored history of interaction events. In this way, the machine’s user interface can adapt to the usage habits of the machine operator. Functionalities and corresponding information elements are hidden from the user interface if the machine operator accesses them only rarely. This information could be stored to an OPERATOR PROFILE so that operators can have a similar user interface presented independent of the machine they are using. However, this kind of approach might be confusing for the inexperienced operator, as the user interface changes dynamically. When using the TASK-BASED UI approach, the machine operator should be offered use of the “classic user interface,” meaning the context-based user interface switching is turned off and operators manually select the view they want.

While designing a TASK-BASED UI one should note that some of the user interface elements are such that they still need to be shared between different task views. For example, global status indicators such as a low oil pressure alarm, error messages, or HMI NOTIFICATIONS should be the same regardless of the active user interface view. Often this is addressed by having a status area that is common for all views in the user interface.

Deciding which items are shown on the screen in each mode for a task is not an easy exercise and it can sometimes be tricky to implement. There is a lot of literature available on this topic. For example, (Welie, 2001) presents some user interface design patterns and tools for task-based user interface design in his Ph.D. thesis. Another Ph.D. thesis (Kersten, 2007) describes a similar task-focused user interface approach that has been validated in three separate field studies.

The ROLE-BASED UI pattern describes an approach in which each user group is created with its own user interface. This way, access to information can be limited according to the role of the person using the machine. Furthermore, extraneous information can be hidden based on the role. While using a TASK-BASED UI approach one should apply COMMON LOOK-AND-FEEL to ensure that all task-based user interface views share the same graphical style and operating principles.

✥ ✥ ✥

A TASK-BASED UI makes it easier for the machine operator to find the information relevant for the current task from the user interface. Because the
user interface always corresponds to the current operating context, usability is improved and the operator can work more efficiently. The task-based user interface has been validated as showing statistically significant increases to operator productivity (Kersten & Murphy, 2007).

A TASK-BASED UI limits the user interface elements that are visible at any one time. This makes it possible to use lower-end hardware to offer a fluent user experience, as fewer elements require updating simultaneously on the screen. Furthermore, this might also affect the load on the message bus (if ONE TO MANY has been applied) as fewer elements require fresh data from the nodes.

A TASK-BASED UI might be somewhat confusing for the inexperienced machine operator because the machine has to be taken into a correct state before certain controls or information can be accessed from the screen. However, even without a TASK-BASED UI all elements might not be visible simultaneously on the screen and would need to be accessed by navigating in the menus.

It might be hard to identify which tasks require their own user interface. It might even be impossible to apply this approach if the functionalities of the machine are such that they can be used simultaneously, e.g., the machine can be driven forward while controlling the boom. If this approach were applied to such a machine, there might be a need to limit the ways in which the machine can be operated.

A grader used to create a flat surface in the grading process at construction sites uses a TASK-BASED UI. There are three main operating modes: driving forward, grading forward, and driving in reverse. The default operating mode is driving. While driving, the display in the HUMAN-MACHINE INTERFACE shows the engine RPM and speed and other information such as fuel level. When the operator presses a button to lower the grade, the control system enters grading mode and the user interface updates automatically. The size of the RPM and speed gauges becomes smaller and video feed from the grade is shown on the screen. In addition, the grade’s cross slope angle is displayed, along with the status of slope balancing automation—whether it is on or off. If the machine operator puts the grader in reverse gear, the operating mode is switched to driving in reverse. In this mode, the machine checks whether the grade is lowered. If it is, an HMI NOTIFICATION is shown to the operator. Otherwise, the user interface updates to reverse driving mode and all the driving-related elements are displayed onscreen. In addition to those controls, a video feed from a reverse camera is shown onscreen.

**Multiple Operating Stations**

...there is a work machine with a HUMAN-MACHINE INTERFACE enabling the operator to use the CONTROL SYSTEM. The work machine has implements or booms attached which are used to carry out tasks. The rest of the machine, called the base machine, is needed to enable the system to move, produce power to the implements, and/or to shelter the operator. The HMI enables control of these implements. The operator has to focus intensely on controlling the machine or implement and is thus subject to a considerable amount of stress. Therefore, adding implements to same machine is limited by the cognitive capabilities of the operator.
The working environment may be such that several implements, such as booms, could be in use at the same time. Substantial cognitive stress may be imposed on the operator if he or she needs to focus on controlling multiple implements simultaneously.

Controlling even just one implement may be a huge cognitive stress for the operator. However, in some cases, several implements could be attached to the base machine to increase its productivity. This usually occurs when the tasks are parallel in nature, not sequential, such as drilling holes for explosives in a mine. Of course, several machines could do the work in parallel, but adding an entire machine just to enable one more implement would be wasteful. Using multiple machines might be impossible anyway because several huge machines wouldn’t fit in a small work area, e.g., a mine. In addition, the machine is quite expensive. In any case, the new machine would need an operator.

Efficiency for certain work can be increased if several operators are used, each of whom focuses on monitoring a given subtask. If an operator is focused on a certain task, she may miss some important details or hazards. Thus, several eyes even on one problem may increase efficiency and safety. One operator may even learn from the other operator how to work in similar way as an apprentice learns from a master. However, safety must be taken into account if several people are working in the same area or on the same task, as different implements may collide with each other.

Depending on the country where the machine is used, the salary of the machine operator varies. In some countries, the salary is high compared to the machine’s acquisition price, and in other countries the salaries are a negligible cost. In the latter case, multiple operators could be using the same machine if it has many work implements. For example, a mining drill could have multiple drills, each of which is operated by its own operator. This could improve productivity. However, because high salaries are an issue in some regions, all work implements should be operable from a single operating station. In other words, the machine should be scalable in terms of simultaneous users.

In some environments, several machines swarming around the work area will increase the risk of collision between machines. Thus, having fewer machines on site will increase safety.

Therefore:

Add a new operating station for each task to be performed with the machine. These operating stations can be identical or specialized for a specific task. This enables several operators to control the machine at the same time.

Create several operating stations by adding multiple HMI buses to the system, each connecting to the system bus. With this connection, these additional HMI buses can send control commands to the system. There are several possible configurations for the operating stations depending on which
kind of machine it is implemented on. If the machine is such that all implements carry out similar tasks, it may be sensible to make all operating stations identical. This kind of setup is shown in Figure 15. Here, every operating station can be paired with a certain implement, which it will control. Usually the HMI bus master takes care of sending messages to the correct implement on the system bus. Even though all implements are separated, it may be necessary for the CONTROL SYSTEM to keep track of the commands and prevent the collision of implements. This requires that enough information on the current positions of the implements is available and enough computing power to calculate the movements. Therefore, usually this is done on a separate node.

![Diagram of two identical operating stations on the same machine](image)

**Figure 15:** Two identical operating stations on the same machine

Putting all controls onto separate buses causes some additional costs, but it enables the controls to be identical to the level of bus identification, meaning that every control device may have the same address and configuration file. This way, there is no need to configure devices differently depending on which operating station they are installed. This makes maintenance easier and speeds up manufacturing in the factory assembly line.

In another approach, the operating stations control the same functionality or implement. Thus, the basic controls are identical. This allows flexibility in switching operators on the fly. If simultaneous control commands are given, some operating stations may have the priority over the others. One approach that is usually feasible in steering and piloting machines is to calculate some average based on the controls, but this usually leads to confusion, as the operators feel that the machine does not respond to their commands and may try to use larger control movements to get the feedback they want.

In some cases multiple operating stations control several different functionalities. In this approach, from each station only a corresponding functionality can be controlled and the controls can vary in each station as different functionalities need different controls. In this case, multiple machine operators should not be able to do contradictory control maneuvers. If these stations share similar functionalities or some functionalities will have an effect on other stations, the functionalities should be exclusive. For example, if one operator is adjusting a parameter affecting the hydraulic pressure to the implement, the same parameter cannot be adjusted from another operating station.

One possible configuration is for the operator to select the task she wants to monitor on the operating station. This is usually feasible if the system controls a process which can be monitored and adjusted only by using the displays and controls at hand. For example, in a factory, the operator may select a certain point of the process she wishes to monitor. The user interface shows all the relevant information and the operator may adjust the parameters for the monitored part of the process. This way, one operator may control the whole process, but usually there is so much going on that the one operator is not enough. Thus, several operators may monitor the process and focus on certain
parts of the process. Configuration options for MULTIPLE OPERATING STATIONS are depicted in Figure 16.

![Figure 16: Several possible configurations for multiple operating stations. From top to bottom: a setup where all stations can control all implements, a setup where each station controls only a certain implement, and a hybrid, where one implement can be controlled from both stations.](image)

The ALTERNATIVE OPERATING STATION pattern also results in a work machine having several operating stations. However, these operating stations are meant for controlling the same task and are mutually exclusive. The other operating station is meant for temporary use—for example, to provide a better view than from the main operating station. The APPLIANCE-PROVIDED UI pattern may be used in conjunction with MULTIPLE OPERATING STATIONS. This way, the implement may configure an operating station for use.

Several implements may be handy in order to provide some kind of redundancy. If an implement breaks down, other implements may still continue working. See the LIMP HOME pattern and the 1+1 REDUNDANCY pattern on utilizing redundancy to continue working in case of malfunction.

A machine can have multiple operators, each focusing on and performing simultaneously their own tasks. This increases the machine’s productivity with a cost increase that is small compared to the price of a new machine. However, in some cases, some properties of the machine, such as mobility and power generation, may restrict productivity compared to a new machine.

If multiple operators are using the machine, collisions of the booms, etc., need to be prevented programmatically or by mechanical design; otherwise the operators must take care to not interfere with one another.

It might be hard to design a system so that a command sent from one station that would affect all other operating stations won’t cause surprising situations to other operators. Thus, this kind of interaction should be minimized. OPERATING MODES may help in separating these concerns, as global commands may be limited to only some specific mode.

The solution makes the system more complex because different operating stations need to exchange information. For example, it must be decided which operating stations show which NOTIFICATIONS. Furthermore, exclusion of
functionalities between operating stations should be implemented, which also increases complexity of the system.

In a paper mill, there are several workstations offering facilities for the operators to monitor and adjust the parameters of the process. Each operating station is easily configurable with the TASK-BASED UI pattern to suit the needs of the operator. For example, if the operator wants to monitor the moistness of the paper mass, she might select the drying screen controls to the operating station. That way, the operating station shows all relevant information on the display and enables the operator to change the control parameters.

**Appliance-Provided UI**

*Also known as Implement-provided UI*

A modern work machine running a CONTROL SYSTEM is typically a very sophisticated one with a lot of computational and physical power. As such, it is able to carry out various tasks if the implements and the equipment allow it. Thus, it is feasible to design the machine so that it enables changing implements and equipment. However, advanced implements can be complex equipment and may have several sensors and actuators. Thus, they need their own CONTROL SYSTEMS to take care of their functionality. For this CONTROL SYSTEM, it is not cost-effective and user-friendly to have them on a separate HUMAN-MACHINE INTERFACE.

A multitude of optional equipment can be attached to a work machine. The optional equipment requires its own controls and displays, but there is not enough space in the cabin for all the additional controls.

During the design phase of the work machine only some of the additional equipment that can be attached to the system is known. Thus, designing a UI for all possible implements known and preconceived is a very difficult, if not straightforwardly impossible, mission. To add to the difficulty, different third-party vendors may have slightly different approaches and devices. Even if preplanning the system for all the future possibilities were possible, in most cases it would be in vain because all the equipment is never connected to the work machine anyway.

A long life cycle also poses challenges, as secondhand machines may be used in ways that the original owner did not think important. Thus, it should be possible to customize the machine with different implements during its life cycle.

Usually, numerous sensors are installed on an appliance. Some of the information produced by them is internal to the appliance only, but most of it is needed on the work machine, too. The operator usually needs to know at least some of the sensor information. For example, in an agricultural setting, the
tractor driver, while using a sowing machine, needs to monitor the mixture and adjust the amount of seeds sown into the soil if the mixture is not optimal. In order to do this, he or she needs to have access to sensor information on the sowing machine. Furthermore, a typical implement needs a huge amount of controls. Thus, having all these controls as physical devices would consume so much cabin space that it is not feasible. In addition, filling the cabin space with controls would inevitably lead to non-ergonomic working positions for certain controls.

As modern implements have several actuators, the amount of hydraulic connectors may be insufficient. Thus, it is easier to feed electric power and let the implement generate its own hydraulic power.

Therefore:

Create a virtual terminal capable of presenting a user interface to the equipment’s own HMI. Create a format for user interface templates and a way to present them on the virtual terminal. Each optional item of equipment then provides its own implementation of its user interface using the specified format. When equipment is attached to the machine it sends its user interface presentation to the virtual terminal. The equipment can present information on the user interface view and receive control commands from the machine.

Design a virtual terminal which can be either a separate physical device, a display in the cabin, or a screen/window/workspace on the cabin PC (see SEPARATE REAL-TIME). The terminal can show UI elements needed by the optional equipment. The UI elements are predefined and specified such that both parties (the control system on the work machine and the implements’ CONTROL SYSTEMS) agree on them. For example, it can be agreed upon that the UI is presented in XML schema (Oksanen et al., 2011). It is usually most sensible to provide a set of view templates and widgets you support on the CONTROL SYSTEM and provide these to the implement vendor. That way, the implement manufacturer may construct its own UI using these components.

When an implement is connected to the work machine, it attaches to the communication bus on the CONTROL SYSTEM. Usually the communication is carried out with a standard connector. Thus, the work machine must provide a connector for interfacing with the bus of the work machine control system. In some cases, it may be possible to attach several implements simultaneously. In the start-up phase, the implement control system informs the work machine control system that it is present and provides the UI for the virtual terminal. The virtual terminal can provide some general controls and input devices, such as physical press buttons, dials, joysticks, touch screens, and other means of control, such as mouse and keyboard. When the connection is established, these controls are mapped to certain commands on the appliance, so when the operator makes adjustments with the UI, the virtual terminal only sends corresponding commands to the appliance control system. Therefore, no business logic needs to be implemented in the work machine control system, and the virtual terminal is only a visualization tool. See Figure 17 for an example of a virtual terminal configuration. The language selection can be done using internationalization information residing in the control system.
In the connection handshake phase, the virtual terminal and the implement share information regarding for which capabilities the terminal has to show information and what kind of HMI devices it can provide. Also, localization and such things are agreed upon, as the work machine has the information about the operator preferences. Similarly, the implement indicates what commands it expects and provides some guidelines regarding how to present this information on the screen. After the connection has been established, the implement and the virtual terminal send HEARTBEAT messages to each other to ensure that the connection link is still working. The same applies when several implements are attached. The implements must have a unique identifier so they can be monitored and configured individually. The display can even be updated such that the implement pushes information to the virtual terminal, which acts as an observer for the messages. Another way to poll the device for changes in the values of the objects shown on the display. Using a VARIABLE MANAGER is one way to implement this.

Security must be taken into account because the appliance connects to the work machine bus. The appliance should not be able to read any sensitive data from the work machine and it should not be able to cause problems in the bus traffic. For example, if the CONTROL SYSTEM on the appliance starts sending error frames to the bus, it should not affect the work machine CONTROL SYSTEM. One way to mitigate this is to install a MESSAGE GATEWAY to filter unnecessary messaging between the work machine and the implement.

As the APPLIANCE-PROVIDED UI needs to be easily usable and feel familiar to the operator, it should implement COMMON LOOK‘N’FEEL across all implements and the actual work machine control system. For example, only a certain button should change the UI language for all devices. To ensure this, some guidelines for designing the UIs should be provided. In addition, the APPLIANCE-PROVIDED UI is one way to implement CONTROL SYSTEM OPTIONS as implements. The difference is that when using the APPLIANCE-PROVIDED UI pattern, the options do not need to be specified during the compile time of the control system; new options can be added during later stages of the product’s life cycle. Also, a TASK-BASED UI can be used to ensure that when a certain implement is connected, even the actual work machine UI shows only information that is relevant to the tasks that the implement enables.
The ISO 11783 (ISO, 2007) standard is based on J1939 and defines a communication protocol which enables agricultural implements to provide UIs for tractors and forestry machines. Its practical implementation is called ISOBUS.

When the APPLIANCE-PROVIDED UI pattern is applied, all implements may be manipulated from one place, namely the virtual terminal. This makes operating them easier, enhancing usability. However, it must be taken into account that for a single button, the meaning may change if the connected appliance changes.

With an appliance-provided UI, the work machine manufacturer does not have to know in advance all possible appliances that might be plugged in to the system during the life cycle of the machine. In addition, different vendors may connect without compliance problems. Testing all possible combinations would be an impossible task.

An APPLIANCE-PROVIDED UI also makes the systems cheaper because implements need not to have separate HMI devices. However, in some cases the appliance may need very specialized controls. In this case it might be impossible to use the virtual terminal, and a separate control device must still be mounted to the work machine.

Information security and safety must also be handled, as unknown devices may be attached to the control system. They should not be able to affect the work machine’s basic functionality.

In a modern agricultural tractor, the CONTROL SYSTEM allows connecting several appliances to the tractor frame. Appliances can use the power takeoff for rotational power, hydraulic connectors for hydraulic power, and attach to the tractor electronically via ISOBUS. During the year, farmers use the tractor with many implements. To get the crop started, they must plow the field to prepare the soil for new growth, then harrow it to pulverize the plow furrows to a usable field, and finally use the sowing machine to plant the seeds and the fertilizer into the soil. A four-furrow reversible plow needs hydraulic power to adjust the main shares and to rotate the whole plow around at the end of the furrow to enable the farmer to start a new furrow. In the tractor cabin, a terminal shows the attack angle of the main shares, the depth of the disc coulters, and the position of the moldboard. The terminal enables the farmer to adjust all four main shares, coulters, and moldboards separately in order to optimize the furrow depth and the form of the furrow. In addition to this, the terminal shows if some share beam has risen due to collision with a stone or similar hard object in the ground. The farmer may also control the hitches via the terminal to raise the whole plow if the tractor is about to become stuck.

HMI Notifications

...there is a CONTROL SYSTEM with a HUMAN-MACHINE INTERFACE. Events occurring in the system are communicated using NOTIFICATIONS. Hundreds or even thousands of NOTIFICATIONS are sent through a bus in the system per minute. Naturally, only a small fraction of these numerous events are such that they require the machine operator’s attention or are of interest to the operator. Some of the notifications are such that the system cannot continue normal
operation without the operator deciding what the system should do next. For example, when functionality fails to complete, the operator must decide whether trying again is required.

Many events occur in a system. Only some of the events are interesting to the operator. Some of these events may require an operator decision in order to continue operation.

When a hazardous event occurs the machine should respond to the situation as fast as possible to ensure safety. However, often a HUMAN-MACHINE INTERFACE is not running in real time and thus has latencies. Therefore, the machine has to react to the event by itself; it cannot wait for human intervention before taking action.

While there are a plethora of events occurring in the system at any given moment, only a small share of them is of interest to the machine operator. Thus, to offer good usability the machine should inform the operator about events that may be interesting. For example, when a functionality with a long execution time, e.g., SELF-TESTS, completes, the operator should be notified. Furthermore, different kinds of events are interesting to the machine operator. Some of them do not require human intervention, while some require a decision from the operator. Therefore, the system should have capabilities to handle different kinds of NOTIFICATIONS in a different way.

Because bus capacity if often limited, NOTIFICATIONS deliver event data in the system in a nonhuman-readable format. Human-readable event information often requires more data to be transferred. However, to be useful for the machine operator, the event data should be represented in a convenient way so that the operator can understand it.

All messages shown to the machine operator should be localizable in order to make them readable and understandable. In other words, the message should use language and units of measurement which are familiar to the operator. Sometimes the machine is used by multiple operators, e.g., a different operator in each shift, so the language and units of measurement should be easily changeable—for example, by switching an OPERATOR PROFILE.

A HUMAN-MACHINE INTERFACE often has indicator lights that can be lit when an event, warning, or fault occurs. However, indicating that an event has happened is not always enough. More information needs to be given to the operator so that she can decide what should be done next. Thus, a textual message about the event should be provided.

Therefore:

Define a subset of notifications that are of interest to the operator.
Create an HMI notification service to display these on the user interface. Some notifications require actions that the system cannot resolve autonomously, and thus require an operator decision. The service should
therefore be able to send a control command to the bus when the operator responds to a specific notification.

Add a notification service to the HMI master node or to a node which offers the GUI for the machine operator. The notification service listens to NOTIFICATIONS on the bus and filters them. Only the notifications which are relevant for the machine operator are shown on the HUMAN-MACHINE INTERFACE. Usually there is a filter configuration file, e.g., an XML file, specifying which notifications are displayed to the operator (see Figure 18). Alternatively, a node containing the notification service can be configured using a VARIABLE MANAGER to listen to only certain notifications on the bus. These notifications are those that would be interesting to the operator, and they can then be shown on the graphical user interface where all received events are displayed.

One typical way of presenting HMI NOTIFICATIONS to the operator is to use pop-up windows on the GUI screen. These pop-ups are then closed by pressing a certain button or in the case of touch screens by tapping on the screen. However, when using pop-up windows to present the notifications, one must be careful not to overuse them as it might become annoying for the operator and reduce their effectiveness. In addition, indicator lights in the dashboard or GUI can be used to remind the operator that e.g., a fault is still active even though the pop-up window is not shown anymore. Similarly, some of the NOTIFICATIONS might be such that if the problem persists, the pop-up is shown again after a certain amount of time to remind the operator that the issue still needs handling.

Typically a pop-up notification window contains three elements: an icon indicating the type of event that occurred, human-readable text describing the event, and possibly values related to the event. If the NOTIFICATION LEVELS pattern has been applied, the icon can be directly mapped to the notification level. For example, Notice level notifications can have exclamation mark with a blue background, Warnings can have an exclamation mark on a yellow background, and Faults can have stop sign on them. The purpose of the icon is to make the notification easily identifiable for the operator. In addition, more severe notifications such as faults may have BEACONS attached to them. When designing HMI NOTIFICATIONS and choosing icons, etc., one should aim for consistency throughout all the notifications. See the COMMON LOOK-AND-FEEL pattern for more details.
Some of the HMI NOTIFICATIONS might be such that they disappear after a while automatically. Some of them might be shown on the screen until the operator acknowledges the notification, but they won’t prevent operating the machine. However, for example, fault level notifications might be such that the machine reacts to them automatically, e.g., by entering SAFE STATE. If this is the case, the HMI notification might still be shown to the operator after the machine has stopped. Furthermore, some HMI NOTIFICATIONS might be coupled with NOTIFICATIONS which have states. For example, a low oil pressure notification might generate an HMI NOTIFICATION, too. Although the operator acknowledges this kind of notification, the oil pressure is still low. Usually this kind of information is shown as a status light or other indicator in addition to the HMI NOTIFICATIONS.

Some HMI NOTIFICATIONS may require the machine operator to make a decision. For example, the notification could contain a question such as “The automatic calibration of the boom leveling sensors failed. Do you want to try again?” In this case, the operator should be able to answer to the question by tapping the screen or by pressing a specific button. On the lower level this means that the notification service must be able to send control commands to the bus. In the case of the preceding example, the notification service should be able to re-initiate the automatic calibration sequence. Furthermore, when designing an HMI notifications service, keep in mind that the machine operator should not be part of any safety-related control loop. The control system should react on the fault notification immediately, even before the operator notices the notification.

The human-readable text for notifications is typically stored in a separate file on the node where the notification service is implemented. This way, the human-readable text does not need to be transferred with the notification message on the bus. Text is fetched from the file using the notification ID which is attached to the notification message (see the NOTIFICATIONS pattern). This also makes it possible to localize the strings quite easily. Sometimes there might be values attached to the message. For example, if the engine temperature has risen too high, the temperature value can be shown to the operator. In some cases, this value might provide additional information to the machine operator so that she can further investigate the situation. Note also that the values shown to the operator may need to be localized as well. For example, the speed or distance values should be localized depending on which unit system (metric or imperial) is used.

In addition to showing the notifications to the operator, the notification service can also provide an event log for the operator or maintenance engineer. The event log enables all events that occurred to be seen at once. Typically the event log is implemented as a separate view in the GUI, so that the operator or maintenance engineer can check which notifications have occurred in the system. This log view can use different granularities depending on the role of the user. For example, the operator might see only notifications that have been shown to her anyway and the maintenance engineer can see all the events that have occurred in the system. To see how different user roles can be identified, see the ROLE-BASED UI pattern. To read more on logging, refer to the NOTIFICATION LOGGING pattern.

One should consider also using the FIVE MINUTES OF NO ESCALATION MESSAGES pattern (Adams et al., 1995) to limit the number of HMI NOTIFICATIONS shown to the operator. In many cases a fault might cause multiple notifications to be shown to the operator, but only the first one or the most important one should be shown. Sometimes, it might be hard to determine which of the messages to show. The DATA STATUS pattern can also be utilized
for deciding what kind of HMI notification to show to the operator. In some cases, a data status value can be used to decide on the notification type.

The machine operator can be informed about events that are relevant for her. In addition, these HMI notifications contain human-readable information about the situation. The machine can still react to the notifications immediately and the response times are not limited by a non-real-time part of the system.

Bus load is not added, as human-readable texts reside on the node where the GUI is running and the messages do not need to be transferred over the bus. Furthermore, localization of the notification messages is easy.

The notification service is the natural place to create a logging mechanism, as the logs can be generated from the notifications and then accessed through the graphical user interface.

It can sometimes be hard to decide which messages should be shown to the machine operator. Furthermore, it might be hard to find the root cause of a failure and show information about it in the HMI notification.

Overusing pop-ups might be annoying for machine operators, especially if they need to acknowledge them. If pop-ups are used heavily, they lose their meaning and the operator will acknowledge them without reading the text in the window. Therefore, pop-up windows should be carefully considered as a means of informing the operator.

In a work machine a separate real-time pattern has been applied to divide the system into two levels: real-time machine control and non-real-time graphical user interface. The system master PC is used to provide the graphical user interface for the machine operator. It is connected to the backbone CAN bus and listens to all messages on the bus. HMI notifications are applied and the notification service is created on the system master. It logs all the bus messages it receives. It has been configured to show a subset of all notifications to the user, and a configuration file lists the shown notifications. Based on the ID of the notification, the notification service also maps a corresponding UI text to the notification. The texts reside in a separate file containing the notification texts as key-value pairs. A separate language file for each language enables easy localization. The desired language file is selected based on a language parameter value, which the operator can adjust from the UI.

Black Box

...there is a control system where diagnostics data is collected. Even though the diagnostics data is mainly used for avoiding unplanned breaks in production, the data may help to determine the cause of malfunction and detect various fault patterns. In addition, a bootstrapper will check the system’s condition before starting the application and self-tests enable testing the system even at runtime. However, critical failures are usually caused by malfunctions that may not have appeared in the system condition before they cause the failure. Additionally, bootstrapper or self-tests cannot provide useful information for service personnel if the diagnostic information is lost. The information might be lost due to a malfunction or because the information
cannot be accessed through the HUMAN-MACHINE INTERFACE or via REMOTE ACCESS because the system is not functioning properly. These factors may prevent reproducing the problem or at least make it harder. In any case, without proper information, analyzing the root cause may be impossible.

When a system malfunctions, the malfunction may prevent access to diagnostic data. It should still be possible to analyze the root cause of a failure afterwards even when the system is in a nonworking condition.

Control systems usually collect various data points from their current condition and save this information for future analyses. For example, log files are a very common way to analyze the situation just before malfunction. This kind of information is usually stored on local storage or it can be accessed through the bus if the system is up and running. If the system has been physically damaged or it cannot be powered up because of the malfunction, the valuable diagnostic data may be permanently lost.

A system usually produces huge amounts of data but only a fraction of it is interesting in terms of failure analysis. If all the data is stored for a long time, the storage space requirements become unfeasible. Thus, only required events should be stored. In addition, very old information is hardly relevant for analysis, so there is no point in storing that information.

Diagnostic and other information can be sent to a remote server if the system has REMOTE ACCESS. However, it usually is not possible to send all the data (see also the DYNAMIC MESSAGE CHANNEL SELECTOR pattern), so various packing techniques are applied and the data is sent as a packet at regular intervals or when the proper remote connection is established. If the system malfunctions before the data is sent, the data may be lost if there is no other way to access it.

Some regulations and standards may require that the information for investigating accidents must be available. Usually accidents cause severe damage to the control units of the system, so any information stored on them may be lost in case of accident.

Therefore:

Add a black box component to the system which keeps records of selected system events. The black box stores these events for a specific amount of time for later inspection. Data in the black box component can be accessed with a separate tool even when the system is not functional.

The system usually produces huge amounts of data that can be used, such as for analyzing the cause of a malfunction. The main idea of the black box is to provide access to the interesting data even if the control system is not functional. The system might be physically damaged, e.g., crashed or burned, or the permanent storage may contain errors because of the malfunction. These kinds of failures prevent normal access to the data. In addition, if the data is accessed via bus or another interface, the system has to be functioning at least to some extent so that the data can be accessed. A black box solves these problems by
storing the information separately from the system and providing an independent way to access the data.

Depending on the system and requirements for the black box, its implementation may vary. In the simplest case, the black box is a USB stick and data collecting application running on the controller. If the system has the SEPARATE_REAL-TIME pattern applied, the application usually resides at the operator level. With this kind of implementation, one can access the data even if the system is malfunctioning and does not start. However, physical damages may likely damage the USB stick at the same time as the main system. In addition, as the system malfunctions, it’s possible that the malfunction might erase or corrupt the contents of the USB memory.

A more secure way to implement the BLACK BOX pattern is to have a physically separate node that collects the events from the communication channel and stores the events in its own permanent storage, such as Flash memory. This kind of node is easier to supply with hard shielding for any physical damages and it may contain an emergency power supply. This type of black box node should have its own interface, such as USB or an Ethernet connection, that can be used to access the data even if the node cannot be accessed using the bus.

As only some events in the system are meaningful for failure analysis, there is no point in storing all system events. If all events were recorded, the storage space requirement would soon become too large. For this reason, the black box can usually be configured using a separate configuration file, typically an XML file. The file defines which events are stored in the black box and for how long they are stored. The latter can be a measure of hours or number of events. The configuration file can also contain other definitions. An example of such a configuration file is shown in Listing 5. The file defines which events the black box will collect. For each event, additional definitions can be given. As already mentioned, the most common one is length of storage, which defines for how long information about the event is stored. Depending on the system, this may be minutes, hours, or even days. In Listing 5, it is 60 minutes for the PLC_VOLTAGE_LOW signal. Other definitions in the configuration file could be, for example, the source of the event, i.e., which node or application has sent the event, and more complex definitions such as the parameter to start the diagnostic data collection after a certain event has occurred. This is demonstrated in the sample definition in Listing 5. This may be useful, for example, with temperature warning events, where collection is started after the event and stopped after the situation is back to normal.

The most well-known example of the black box is a flight recorder (IRIG, 2002), which is used for recording significant parameters of the plane. Similar devices are also available for trains (train event recorder) (Railway Safety, 2002) and other vehicles such as vessels (voyage data recorder) and cars (event data recorder). Recorder data is analyzed when investigating aviation accidents and incidents.
A black box provides access to the meaningful data even when the system is not in a functional state. It stores the events that can be used to identify the root cause of the malfunction. However, a BLACK BOX is usually an additional mechanism; the main sources for analyzing programming errors are the system logs and diagnostic information. Because of size limitations, the black box can contain only the most crucial data that is used in case of a critical accident.

Sometimes it may be difficult to select which events are relevant enough to warrant storage. Because storage space is always limited, some crucial information that could help to solve the problem may have been left out by accident. This information may be impossible to retrieve afterwards.

A BLACK BOX increases manufacturing costs and needs additional space. Conversely, information from the black box can be used to determine if an accident is caused by the operator and not because of a bug in the control system. This information may be essential when considering compensation for damages.

Privacy issues may arise because the black box stores information about how and when the operator carries out his tasks even though this kind of information is not stored in other ways. Tachographs are good examples of this.
A train has a train event recorder that collects information from the train control system. The recorder is shielded so that post-accident investigations are possible even if the train crashes. The recorder has a Flash memory, which is used to store certain status information about the train. Such information includes the current speed and location of the train as well as the status of the control system and user controls. For example, brake activations, door open/closed signals, status of the safety systems, and speed settings are stored. With this information, it is possible to know, for example, if a door has been accessed just before the incident. Black box information is also sent to the central control room of the railway network when the train is serviced.

**M2M Communication**

...there is a fleet of machines and each of them has a CONTROL SYSTEM. The machines in the fleet can communicate with each other through the centralized server of the FLEET MANAGEMENT system. While the FLEET MANAGEMENT is aware of the status of the fleet and can inform machines about the status of other machines, the machines do not communicate with each other. However, sometimes a machine needs to exchange information with other machines to be able to carry out tasks automatically. For example, when two automatic warehouse forklifts arrive at the intersection at the same time, they could negotiate which one of them goes first. This kind of communication could be carried out through the centralized server, but in many cases it would be too slow because the situation in the field is constantly changing. For example, in the case of the warehouse forklifts, they could inform the FLEET MANAGEMENT server about their route and other forklifts could ask for the routes of other forklifts, but this would be cumbersome and slow. In addition, the routes of the forklifts may have already changed because of the dynamic nature of routing—each route might depend on the choices other forklifts will make.

A number of machines work in the same fast-paced environment. Machines may need to share information with each other to ensure fluent operation or to warn other machines. A centralized communication mechanism would be too slow, or does not exist.

There are multiple machines in a fleet and they are working in the same environment. To achieve their goal or to optimize the process of achieving it, machines need to communicate with each other. Alternatively, the machines may need to communicate in order to be able to achieve a common goal.

Mediation in communication always adds latency to the messaging. The communication latency would increase too much if the communication took place through an intermediary such as a FLEET MANAGEMENT server. If the latency in the communication is too long, the volatile environment might have already changed and the context of the communication might have already changed. For example, the machines may not be in each other’s proximity anymore. Using an intermediary requires that the machines publish all information to the intermediary, which then decides which information needs to
be sent to the other end of the communication. However, in this approach the amount of transferred data would soon become too large and the throughput of the communication channel would be compromised.

Sometimes the infrastructure for long-distance wireless communication does not exist and thus prevents communication to the FLEET MANAGEMENT server. For example, in underground mines the communication possibilities are very limited unless WLAN or a similar infrastructure is built. There might also be other devices in the area disrupting the long-distance wireless communications, e.g., some other devices occupy the frequency that would be normally allocated for WLANs. In these cases, the machines still might be able to bridge traffic from machine to machine.

In a large fleet, the machines might be physically located far away from each other. However, often the machines are interested only in local information; events occurring miles away are not relevant to the machine. If all events on a global scale are sent to all members of the fleet, the communication channel might be congested. Furthermore, some machines might not be connected to the FLEET MANAGEMENT server, though they are able to communicate with other machines.

Therefore:

Allow peer-to-peer communication between machines by adding a client and server to each machine. Using this communication, a machine can send predefined messages that contain data such as its location and current environment information.

Add a communication handler to the system that can act as a client or a server depending on the situation. This is necessary if the M2M COMMUNICATION requires a significantly different approach from REMOTE ACCESS. If the intention is to communicate with the machines in close proximity, some additional hardware, e.g., a separate node, might be required to establish the close-range communications—using, for example, WLAN, Bluetooth, Zigbee, etc. Often this kind of connectivity is referred to as an M2M area network. An M2M area network refers to any network technology which is capable of providing physical connectivity and addressing between machines or devices which are connected to the same M2M area network or allowing M2M devices or machines to gain access to a public network via router or a gateway (Burkholder et al, 2012).

Define the requirements for the M2M COMMUNICATION, as the solution is highly dependent on the individual case and the requirements set for the communication. For example, (Schattenberg et al, 2012) describes what kinds of requirements there are for M2M COMMUNICATION in an environment where it is used to increase the accuracy of relative positioning of the machines in a fleet. Generally, consider at least the following aspects:

- Number of communicating parties
- Level of acceptable delay in end-to-end communication
- Interactivity requirements
- Data volumes
- Data exchange frequency
- Whether the communication is initiated by the server or the client
- Processing capabilities of the communication module (or node)
- Whether the communication needs a facilitator
Depending on the requirements choose your approach to implement M2M COMMUNICATION. Decide if the communication infrastructure should be always available or established ad hoc. If the communication channel must be always available, you can use the PUBLISH-SUBSCRIBE architecture (see e.g., (Birman & Joseph, 1987)) to realize M2M COMMUNICATION. If a ready-made solution is preferred, one can consider using a Data Distribution System (OMG, 2007) for inter-machine communication using the PUBLISH-SUBSCRIBE approach.

If an ad hoc network is preferred, a mechanism must be designed to detect when another machine has entered the proximity of a given machine. For example, machines can detect each other’s networks and start communicating. However, establishing this kind of ad hoc network poorly suits an application that requires fast response times, e.g., detecting other machines in cross-sections of the working area. If ad hoc networks are detected nearby and the situation potentially requires a fast response, it might be a good idea to stop the machine until the connection is established, as the handshake mechanism might take some time. This way, the machine can negotiate with the other machines regarding what should be done next and the situation does not escalate further.

Often the biggest obstacle for M2M COMMUNICATION is that the communication should be fast and a large amount of data must be transmitted in a relatively short time over a quite limited communication channel. Therefore, one should limit the amount of data transferred in M2M fashion to only the data that is absolutely required by the other party. One could use the DYNAMIC MESSAGE CHANNEL SELECTOR pattern to decide what data is transmitted. For the same reasons, acting as an intermediary in the communication is not often possible, as the capabilities of the communication channel are limited.

Sometimes the communication requires that one machine is given the rights to make decisions for the other machines. If there are permanently installed machines taking part in the communication, then these have the right to make decisions. Consider the example shown in Figure 19. Two buses are approaching an intersection and they communicate with the traffic lights wirelessly. The traffic light is acting as a server for the buses. The bus could instruct the traffic light to change to green to give priority to public transport. However, the traffic light is the machine that makes the decision, as it needs to consider the current traffic situation and other buses approaching the intersection. The general principle here is that the final call should be given to the machine which has the most information to optimally handle the situation. The decision-making rights can be assigned to a certain member of the fleet during design time or the machines can negotiate that dynamically. In the latter case, the server, for example, could decide which is the decision maker. At the architecture level, this means that the machines given the rights to make the decision act usually as a server. However, in the previous example, the traffic light could still act as a client for traffic lights from another intersection.

For lower-level M2M protocols, for example, one should refer to the Eclipse foundation’s open source implementations of standard IoT (Internet of Things) and M2M protocols (Eclipse, 2014). If multiple communication channels could be used, one can use the DYNAMIC MESSAGE CHANNEL SELECTOR pattern to select the most suitable for the situation. The topic of machine-to-machine communication is vast, and this pattern just scratches the surface. Entire books have been written about M2M COMMUNICATION, see e.g., (Holler, et al, 2014).
Machines can communicate directly with each other without having one central server that could potentially form a single point of failure and a bottleneck. Furthermore, communication directly between the machines has smaller latencies than communication through a central server, as there is no intermediary in the communication.

Because machines can communicate with each other, they can negotiate which action should be taken. This makes it possible to MINIMIZE HUMAN INTERVENTION (Hanmer, 2007) in optimization of fleet's operation. A human is not needed to make the decisions because the machines can make decisions by themselves.

M2M COMMUNICATION typically takes place in an ad hoc network when the machines are in close range. Close-range wireless communications does not require as much sending power from the transmitter, so energy can be conserved. In machines using battery packs as their power source, this enhances the battery life and the machine can potentially operate for longer periods between charging breaks.

M2M COMMUNICATION, which is essentially P2P communication, is more complex than centralized communication with a FLEET MANAGEMENT server. Thus, implementation is more demanding and prone to programming errors. In addition, communication can take place ad hoc, which is prone to errors such as losing connection with the other communicating party. This kind of environment requires that the machines can continue the communication and work from the point at which the error occurred. Ad hoc communication also requires handshake mechanisms that can be slow, thus limiting its usage in situations where fast response times are required and stopping the machine is not an option.

The machine should have enough spare processing power to handle the communication caused by M2M COMMUNICATION. Sometimes there might be multiple parties communicating with each other; or if the traffic needs to be intermediated through the machine, the load might increase significantly.

Safety-critical communication should not take the place of M2M COMMUNICATION because it is often not a trustworthy means of communication. Thus, M2M COMMUNICATION is mostly suitable for services offering additional value.
One should take care of information security in M2M COMMUNICATION, as it opens a new attack vector to the machine. A similar mechanism to that used in oMANAGEMENT server is providing information about machines which are part of the fleet, this information could be utilized to create a WHITELISTING FIREWALL (Bonilla-Villareal, 2013).

A combine harvester and agricultural tractor use M2M COMMUNICATION to communicate with each other. When the operator starts the combine harvester she establishes connection to the tractor, which is in the near vicinity of the harvester. The tractor acts as a server and the combine harvester connects to that. Once the connection is established, the harvester sets the tractor to “follow me” operating mode. The harvester sends a set of parameters related to this mode to the tractor. These parameters include, for example, distance values to the harvester, and they tell the harvester how closely it should follow the tractor and indicate the relative position of the harvester to the tractor. Once these parameters are set, the tractor will listen to messages from the harvester. These messages contain, for example, location updates from the harvester. When the operator starts to drive the harvester in the field, the tractor is informed about the location of the harvester and will follow it, trying to keep the location offset constant. In this way, the combine harvester can cut the grains and directly empty its load to the cart of the tractor. If for some reason the machines drift too far apart or the connection is lost, the tractor will stop and enter SAFE STATE.

Opportunistic Delegation

...there is a machine with a CONTROL SYSTEM. The control system is connected to a fleet of machines using a FLEET MANAGEMENT application. The machines in the fleet are performing tasks and multiple machines are capable of carrying out the same tasks in the fleet. The tasks that the machines are performing are emergent in nature, so the actual moment the task needs to be performed is not known in advance. The tasks emerge outside the control of the fleet management system, so they cannot be pre-allocated. For example, a fleet of loaders is working on a sandpit and the loaders are assigned loading tasks from the fleet management system whenever trucks arrive to the pit. However, the location and status of the machines in the fleet is constantly changing. For example, a machine might be busy with a task and the moment it will be idle again is hard to predict. Furthermore, the machine might have just undertaken a task and the fleet management application is not yet aware of it. Thus, it is very hard to say which of the machines would be an optimal choice to perform the task. The fleet management application could poll the system, but the situation might change while gathering the information from all machines in the fleet.

Fleet management needs to delegate dynamically emerging tasks for fleet members. The states of fleet members are constantly in flux. The fleet management application must determine the optimal fleet member to carry
out a specific task, but while doing so the state of the fleet might change. Increasing the frequency of state updates from fleet members does not help, as the communication delay is too long.

There are multiple machines in the fleet with the same capabilities. There might be machines of different types as a part of the fleet, but there are multiple similar machines, too. For example, machine type A is able to perform tasks of kind A and machine type B can carry out B type tasks. For example, a fleet could consist of three A type machines and of two machines of type B. Therefore, the tasks of the fleet are not pre_allocated to any specific machine because multiple machines are able carry out the task. In other words, the tasks are dynamically allocated to the machines as they emerge.

The machines may have accepted another task when a new task emerges and thus might not be able to perform the task. In addition, the machines may have moved from their location and thus might not be the optimal choice for carrying out the task anymore.

Tasks are emerging relatively fast and the execution of the task might take a long time. Thus, communication between the machine and the FLEET MANAGEMENT server becomes a bottleneck.

The states of the machines in the fleet are constantly in transition. In addition, communication between the fleet management server and the machines is relatively slow. If the fleet management requested the state of each machine in the fleet to determine the optimal machine to perform the task, the state of the fleet would have changed by the time all machines have responded to the request. Thus, the fleet management cannot query the state of the machines to decide which machine will carry out the task. However, the machines send periodic status updates to the fleet management server.

To optimize the efficiency and productivity of the fleet, the tasks need to be allocated to the most suitable machine. For example, an elevator call should be assigned to the elevator that has room and is the first one to pass the floor from where the call originated. This way, the order of the already emerged tasks can be optimized for the whole. Optimization could mean optimizing the order for minimal energy consumption or for shortest waiting time. Therefore, this means also that the oldest task request does not need to be served first if a more optimal order has emerged.

Therefore:

Create a task delegator service for the fleet management system. This delegator allocates a task to a fleet member based on the best guess of the fleet member’s state without separately querying whether it is able to do the task or not. If the fleet member can do the task, it takes ownership of the task and informs the delegator. If the fleet member cannot do the task, it notifies the delegator, which reallocates the task to another fleet member.

Describe tasks using a common format that both the fleet management server and the fleet management application on board the machine can understand (see the FLEET MANAGEMENT pattern). When a task emerges, the fleet management server is notified and it creates a task that is given to a task delegator. The task delegator is responsible for selecting suitable machine from the fleet to perform the task. The machines send their status information to the fleet management server and the delegator uses this information. However, the information might not be up-to-date because the situation in the field might have changed. The
delegator component can also try to forecast the movements of the machines and changes in the status of the machines.

The task delegator consists of an algorithm that takes the status information of each machine in the fleet into account and tries to optimize the fleet’s performance as a whole and minimize task completion time. Once the task delegator has determined the optimal machine to perform the task, it sends the task information to the machine without negotiating with the machine if it is available for the task or not. The fleet management client decides whether the machine takes the task or not. The machine may refuse to perform the task if its status has changed while the task has been in allocation. If the machine refuses to perform the task, then the task is sent back to the fleet management server where it is given to the task delegator component again.

The task delegator can also keep track of the delegated tasks, so it won’t assign too many tasks for a single machine, as that would increase the waiting time of the tasks. Furthermore, the task delegator can also try to optimize the situation if the task consists of multiple subtasks that need to be carried by different machines. For example, an ore loading task might need to be postponed until a free vehicle can take care of the transportation.

Figure 20 illustrates the structure of OPPORTUNISTIC DELEGATION. Emerging tasks are buffered in queue (see MESSAGE QUEUE) using the common format used to describe the tasks. The queue is used to level the peaks as the number of tasks grows. The task delegator reads a task from the queue and uses the allocation algorithm to allocate the task to a machine. The INTERCHANGEABLE ALGORITHM pattern can be applied to allow the selection of the algorithm from multiple options. For example, there might be different allocation algorithms to optimize different things—one algorithm optimizes energy usage while another optimizes the time spent to carry out the tasks. When the INTERCHANGEABLE ALGORITHM is applied, a context component is added. A context component is aware of the fleet’s status and can decide which allocation algorithm to use. For example, one algorithm might be used when some machines in the fleet are unavailable, while another is used in normal conditions. If there is no context component, the task delegator uses the fleet’s status information directly to determine the optimal machine for the task.

![Figure 20: A task delegator can utilize multiple allocation algorithms by applying the INTERCHANGEABLE ALGORITHM pattern.](image)

There might be also different allocation strategies during request peaks, etc. In general, the optimization algorithm should be fast. Although accuracy in machine selection can often be optimized by using more time, it is often
unnecessary and good enough results should be used, as the fleet’s status is in constant transition; and the longer the algorithm will take, the higher the chance that the allocation will fail due to changed conditions. This is true especially when the tasks are relatively short. Once the allocation result is received from the algorithm, the task is sent to the machine, as already discussed. If the machine breaks down the task might be lost, if the task delegator is no longer aware of the task’s existence. Usually this is not a problem, but if it is important that all tasks are always carried out, the fleet management server needs to keep track of them.

If the machine does not respond to the task allocation, it might mean that the connection to the machine is lost. In this case, the task is delegated again by the task delegator. If no machines are free at the moment of task allocation, the task is allocated to the machine most likely to finish its tasks first.

**OPPORTUNISTIC DELEGATION** enables different task allocation strategies to be used depending on the current status of the fleet. If some machines are unavailable for some reason—for example, due to yearly maintenance—different allocation strategies can be used. Furthermore, the solution can be extended so that idle time is used to guess which tasks will emerge next. This makes it possible to prepare the fleet for a new emerging task. If emerging tasks are stored, the system can learn patterns about how these tasks emerge, enabling it to predict which tasks are likely to emerge next. In addition, a new component that creates tasks when the fleet is otherwise idle can be added. This way, these generated tasks can be used, for example, to send the machines to a location where the next task is likely to emerge. For example, in a mine, loaders could be moved to a loading area when idle by generating a task for them.

The **EARLY WARNING** pattern can be applied to task queue to detect usage peaks if necessary.

❖ ❖ ❖

Tasks can be allocated to the machines of the fleet in an environment where the state of the fleet is in constant transition. While the task allocation may occasionally fail, the delays are not significant because the allocation process is rather fast. Furthermore, cumbersome and slow negotiation regarding which fleet member will carry out the task is avoided.

A delegator component can be used to gather task allocation data. Based on this data, one can analyze usage peaks and optimize the fleet’s behavior.

When the task is not accepted by the fleet management application of the machine, it needs to be queued again and reallocated. This changes the order of the tasks, so this solution is not suitable for environments where the tasks need to be performed in the order they emerge. Furthermore, reallocation takes some time, especially multiple tasks are emerging at the same time.

**OPPORTUNISTIC DELEGATION** is difficult and sometimes even impossible to implement if the machines communicate in peer-to-peer fashion, i.e., M2M COMMUNICATION is applied to the fleet management. In the case of P2P fleet management, there might not be a central server to carry out the task delegation. However, ready-made solutions such as DDS (OMG, 2007) could be used to solve the lack of central server, although the machines would need to take care of task delegation if there is no dedicated server for that.

❖ ❖ ❖
An elevator group consisting of six elevator cars is controlled by a group controller. An elevator car can move six meters a second, so when approaching the floor it needs to start decelerating from a safe distance so that the passengers are not influenced by an overly high g-force. Thus, the deceleration distance to the floor can be rather long. A group controller can be seen as a fleet management application for elevator cars. Once the passenger presses the elevator call button, a task is generated in the group control application. The group controller uses OPPORTUNISTIC DELEGATION to allocate tasks for the elevator cars. Even if the group controller can communicate with each of the cars and ask for its position in the elevator shaft, the communication latencies are too long; before the group controller could decide to which elevator the call should be allocated, the situation would have already changed and the cars might have passed multiple floors. Thus, the group controller uses an approximation of the positions of all elevators in the shaft. From time to time elevators update this information by sending floor information to the group controller. This information can be sent every time the elevator stops on a floor or a call button inside the elevator car is pressed. Based on the approximate information, the group controller allocates calls for the elevator cars. If the car has already passed the safe deceleration distance to the floor from which the call came, it will refuse to answer the call. Then the group controller is notified and the call is reallocated to another elevator car.

System Adapter *

Also known as Device Proxy

...there is a CONTROL SYSTEM that needs to function as a part of a FLEET MANAGEMENT system. The fleet management application sends tasks and production requests to the control system on the machine. The machine also sends production reports, etc., back to the fleet management server. However, the machine might have an information model that’s incompatible with the FLEET MANAGEMENT system the customer uses and thus cannot communicate with the server. Sometimes, especially in the case of autonomous machines, the fleet management may send direct commands to the machine. These commands may vary between machine models and vendors. Customizing either the control system or the fleet management server for each customer separately would be expensive, time-consuming, and error-prone. A customer would still want to have all machines as a part of the fleet management system and allocate tasks and receive production reports from the machine.

The fleet consists of all kinds of machines from various vendors carrying out several tasks. Customizing a fleet management system for each machine is expensive and time-consuming. From the fleet management’s perspective, however, all machines could be managed in a similar way.

To create an efficient fleet management system all machines need to be handled uniformly. Uniform handling makes development of the fleet
management system easier, as the developer does not need to take care of the
details of providing information to a single machine. Conversely, it is in the best
interests of a machine manufacturer to have support for its machine in as many
fleet management systems as possible. Thus, the machine manufacturer should
ensure that the machine is compatible with different kinds of fleet management
systems. A machine typically has a model number that describes the data it
handles. This model might not be directly compatible with the FLEET
MANAGEMENT system, as models differ. Furthermore, if the fleet management
system sends commands to an autonomous machine, the control commands
must be understood by the machine. Unfortunately, some machines might not
support all commands required by the fleet management system. In addition,
some machine models or vendors may offer commands that the fleet
management cannot take advantage of. To tackle these problems the control
system on the machine should be easily modifiable to make it compatible with
the fleet management server.

During its long life cycle the machine can be used in multiple work sites and
by multiple owners. Thus, the machine may need to interoperate with various
fleet management servers or ERP systems if the work sites use different vendors
for those systems. The machine should be able to communicate with these
servers independent of server version and vendor. Of course, the on-board fleet
management application could be updated or customized separately for each
fleet management system but in the long run it might be impossible.
Furthermore, making changes to the on-board fleet management application
would be expensive and time-consuming. Therefore, using a different fleet
management system should not cause a lot of changes to the control system or to
the fleet management application.

The machine needs to be able to communicate with various machines from
different vendors. The communication might take place through the FLEET
MANAGEMENT server or directly through M2M COMMUNICATION. In both cases,
the machine needs to be able to communicate and co-operate with other
machines. These other machines might even be such that they were non-existent
at the time the control system was designed. While it might not be possible to
prepare for all future communication needs, the modifications required to make
a machine interoperable with new machines need to be minimized in order to
minimize costs.

If the machine control system or the fleet management server were
customized to be compatible with each other, problems would arise when either
system were updated (see UPDATEABLE SOFTWARE). In the case of a software
update, the offered services in the fleet management system or the machine are
likely to change and modifications would be required. Because the fleet
management (or machine) vendor probably has a lot of customers, each update
would require a vast amount of manual work, as each customized system would
probably need modifications. In addition, this kind of manual customization
approach easily leads to programming errors.

Therefore:

Create a system adapter that converts fleet management commands to a
format that the system can use. Similarly, the adapter converts production
data from machines to a format conforming to the fleet management
application’s information model. The system adapter becomes a client for
the fleet management server, wrapping the actual system.
As the fleet management application on the control system is the bridge from the control system to the fleet management system, it is wrapped with the system adapter. The server of the fleet management system sends work plans (or in the case of autonomous system, commands) to the machine through the system adapter. The system adapter offers an interface for the server to use, converting the data to a format that the machine can process. Similarly, the system adapter gathers information from the machine and converts it to a proper format for sending back to the fleet management system’s server.

Depending on the case, the system adapter can be implemented to the CONTROL SYSTEM of the machine or as a part of the fleet management server. In the latter case, the adapter can also abstract the used communication method. This means that the communication between the machine and the fleet management is implemented behind the system adapter interface. This way, the fleet management does not need to take care of the communication method the machine is using. In some cases, it might be rational to use a SYSTEM ADAPTER at both ends: at the fleet management server and at the machine.

Figure 21 illustrates this. There is a system adapter on the fleet management server. Machine 1 has a CONTROL SYSTEM that is provided by the same manufacturer as the FLEET MANAGEMENT system and thus machine 1 has a compatible system adapter. Machine 2 is from another vendor so the FLEET MANAGEMENT system provider has implemented a system adapter for it. Additionally, the server has its own system adapter. This offers some flexibility, as if there are changes in the selected communication channel technology, the changes are always limited only to the system adapter part of the server. Furthermore, other machine vendors might be able to use the fleet management server with their machines, as they can tailor the server’s system adapter.

![Diagram of fleet management system](image)

Figure 21: Sometimes a SYSTEM ADAPTER can be used at both ends: at the fleet management server and at each of the machines.

When a system adapter is implemented on the server side, the system adapter can be implemented as a remote proxy (see PROXY (Gamma et al., 1994)). Figure 22 illustrates this. There is an instance of a machine adapter proxy on the server for each machine that the server controls. The machines are accessed only via proxy. The system adapter converts the requests from the server to the format that the actual machine can process and vice versa. The system adapter can be implemented on the control system of the machine too. However, this approach might be challenging, as the control system might not offer a means to implement the system adapter.
A **system adapter** can be seen as a **hardware abstraction layer** which abstracts the whole machine from the fleet management server. For lower-level hardware abstractions one should refer to **hardware abstraction layer**. M2M **communication** is easier to implement if the control system uses the **system adapter** pattern, as the conversion of the information models is already implemented. **Diagnostics** systems may also utilize a similar mechanism as described in this pattern to make the **diagnostics** system compatible with various machine models.

The system adapter needs to be separately implemented for each machine using a different information model. The implementation should just conform to the same interface description of the system adapter. Sometimes a simple command from fleet management may need to be divided into multiple commands in the system because the machine does not support the command directly. For example, a modern model of a warehouse forklift may support a command “fetch item from shelf X from position Y,” whereas an older model would require several commands to carry out this task, such as “move to shelf X,” then another command to “pick up item from position Y from shelf X,” and finally a command to “return to the loading area.” When using a system adapter, changes to the fleet management or the control system on a machine implies modifications only to the system adapter, not to the rest of the control system or fleet management system.

A system adapter improves interoperability of the machine, as it can function as a part of the fleet which has machines from various vendors and various machine types. Furthermore, when the fleet management server is updated, changes are required only to the system adapter, not to the actual control system. Also, if the machine has **updateable software** and is updated, the possibly changed functionality only affects the system adapter, not the fleet management server itself. In other words, the system adapter makes the machine more adaptable to changes in fleet management environment. However, when using a system adapter, the vendor-specific functionalities and commands cannot be used even though they might be more efficient.

A system adapter adds a layer of indirection between the machine and fleet management. This slows down the communication between the machine and the fleet management server. In addition, the conversion to a compatible format is sometimes resource intensive and may take some time.

From the fleet management server’s point of view, the system adapter makes the fleet management system extensible because new machine types and machines from various vendors can be used as part of the fleet. Therefore, in one sense, the system adapter makes fleet management scalable in terms of adding...
new machines to the system. In addition, the system adapter can be used to stub
the actual machines in the testing phase of the fleet management server, thus
making system testing easier.

A system adapter might be difficult to implement in some cases. For
example, if a command or information sent from the fleet management server
maps to multiple commands or to multiple data items, the mapping might be
hard to design and implement. In addition, sometimes a machine might not
support a feature that the fleet management needs, so the system adapter needs
to fake that functionality by using the capabilities and features the machine has
to offer.


An elevator system uses a so-called group control application to control two
elevator cars in the same building. A passenger on the fifth floor wants to travel
to the ground level and exit the building. When a passenger calls an elevator, the
call is processed in the group control, which then decides which elevator car is
sent to the floor where the passenger is waiting. In other words, a group control
application is a kind of fleet management system optimizing the usage of
elevator cars and minimizing extraneous movements, thus saving electric power.
However, the two elevator cars in the building are from different vendors. One
of the cars has been renewed and provided by another manufacturer. The cars
use different commands. The modern elevator car can be commanded to go
directly to the fifth floor to fetch the passenger. However, the older car needs a
separate command for each floor. For example, if this elevator car is requested
to go from the second floor to the fifth floor, three “go up” commands need to
be sent. In addition, a command to open the door at the fifth floor needs to be
sent separately. Thus, the group control uses a SYSTEM ADAPTER for both
elevator car types. These adapters are implemented by the group control
provider. This enables the group control to command and control all cars
uniformly regardless of the elevator car’s vendor or software version. When the
older elevator car is sent to the passenger on the fifth floor, the fleet
management needs to send only one command (as in case of the modern car).
The system adapter maps this single command to the commands that would
perform the same task with the older elevator car.

**Configuration Parameter Versions**

...there is a CONTROL SYSTEM, where adjustable and/or varying
characteristics of the system are represented and stored as PARAMETERS.
However, as it evolves, new adjustable variables for devices may be introduced
into the system as new devices are needed. Conversely, some other adjustments
may become obsolete or the old values may be unusable in a new device. In
some cases, the adjustable variable may change such that the data type of a
parameter may have to be redesigned. For example, additional precision in a
new generation of an old device may require representing boundaries of
measurements as real numbers instead of integers. Thus, as a new version of the
control system application is released with UPDATEABLE SOFTWARE, the
parameter set may be different from the previous version. Because the CONTROL
SYSTEM consists of several nodes, different nodes might have different
expectations about what PARAMETERS are in use in the system. Even if the
existence of a certain parameter is agreed upon, their types may be
comprehended differently in separate nodes.
A control application update may change parameter values, introduce new parameters, or change parameter types. During the update, the old parameter values should not be overwritten, as the operator might have adjusted them. However, it should be possible to overwrite the old parameter values when absolutely necessary, such as if their type changes.

After software updates, the user would like to preserve as many PARAMETERS as possible, as it may be a tedious task to set them again manually. In addition, in some cases, losing parameter values as a side-effect of software updating would lead to inefficient or even dangerous actions. For example, if the movement limits of an actuator arm are set with parameters, parameters might be reset so that the actuator arm moves to a position it wasn’t previously allowed to enter. This may surprise the operator and cause a hazard.

It is almost impossible to predict how a system evolves in terms of parametrizable values, so over time there will be situations in which the old PARAMETERS are inadequate to describe all adjustable variables in the system. Therefore, introducing new PARAMETERS is inevitable in systems with a long life cycle, causing changes in the parameter storage structure or file format wherever the PARAMETERS are stored.

Therefore:

Store the parameters separately from the application in the update bundle for each unit. Tag the parameter set for the unit with a version number that is distinct from the application version. The updated parameter set contains only new parameters or changed values for old parameters. This allows the old parameter values to be retained over the update.

Store all parameters as a file, which is separate from the application. As new versions of applications are updated with update bundles, it is natural to include the parameter file with this bundle. The parameter set may be stored—for example, as an XML file or an image of memory blocks—in a persistent memory, such as flash memory of the controller. Whenever new parameters are introduced, changed, or removed, a new parameter bundle file is created. This file reflects only relevant changes. The parameter bundle should also include a version number, which describes the version of the actual contents of the parameter set. The parameter set version can be stored as a parameter itself—for example, the first parameter of the bundle would describe to which software versions the rest of the parameters applied.

If parameters are introduced or removed or their structure/type changes, a new version of the storage format is needed. Because the parameters will change, there may be different versions of the storage format itself. The storage format can also be stored as a parameter. From the version number, the application can deduce the correct format and version with which it will need to work. It is the application’s responsibility to use the correct parameter version and the correct parameter storage format in order to operate properly.
The new parameter set should only include those parameters that are new or marked obsolete, to clear them or set them to a default value. In addition, those parameters that have new values or types should be included. The update bundle should hold all relevant parameter set files for the older versions, so that it is possible to build a certain version needed by the actual application installed on the node. In the example shown in Figure 23, the actual parameters are merged from several sources. In the first version, there are two parameters, one for engine and boom each. Then a new parameter for the engine is added in the second version. In the final version, one parameter is overwritten with a new default value and an older parameter is removed.

![Figure 23: Example of a parameter file which consists of three separate versions](image)

If no clear mapping exists from the old parameter to the new parameter—for example, a device has a wholly different type for the PARAMETERS or the mapping cannot be presented as a simple function—it may be mandatory to overwrite the parameter with the default value on the bundle. The system should clearly indicate this and ask the user to set the parameters immediately, so that the system is not used with wrong parameters, leading to inefficient or dangerous operation. One way to implement this is to have a separate OPERATING MODE for updating and use a BEACON to inform the user.

Parameter set structure, their format, and the number of parameters may be updated in conjunction with preserving the old parameters. This allows the user to use updated versions of the applications with minimum hassle.

However, if new parameter set versions are created carelessly; the update bundle may become huge, as all supported versions for the parameters should be included in the update bundle. If the parameter versioning becomes too extensive, it may pose a problem for maintenance and assembly personnel.

In a container handler crane, there are three controllers controlling the engine, the boom, and the frame. The crane is used in a harbor in a Mediterranean climate, but after several years of service it is sold as a secondhand unit to an arctic harbor. To accommodate the vast temperature changes, the boom handler unit is equipped with thermal expansion compensation. Thus, an additional outside temperature sensor is calibrated with several parameters to frame and boom movements. These parameters are only part of the arctic software set and are applied with a software update when the boom is resold. This way, the parameter bundle with the update bundle will include an XML file with new default parameters for the boom and frame. The engine unit does not care about the outside temperature, as it has its own sensor for the intake air temperature. Thus, the engine parameters are not affected by
the software update, but the new frame and boom software versions can use the newly introduced parameters correctly.
References


Gascon, 2008  

Hanmer, 2007  

Holler, 2014  

IEEE, 2011  
IEEE 802.15.4-2011 Standard for Information Technology—Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements. Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR_WPANs). 2011.

IEEE, 2013  

IRIG, 2012  

ISO, 1994  

ISO, 1996  

ISO, 2007  

ISO, 2011  

Jaatun et al, 2008  

Kersten &  
Kersten, M.; Murphy,G.C. “Using task context to improve programmer


Zigbee Alliance, 2014