Open-source run time system for component-based implementation of automation solutions with Java (www.elbfisch.org)

What is Elbfisch?

Embedded PC systems are increasingly being implemented in automation technology, encouraging the design of object-orientated software development processes, a step which is long overdue.

The development environments offered in the rather conservative market of industrial automation technology still lack, or offer only in rudimentary form, properties of object-orientated paradigms now common in IT, such as:

- the encapsulation of data and functions
- the reusability of components
- inheritance hierarchies

and even vital basic functions such as:

- dynamic memory management, and
- protection from unauthorised access to memory areas

This situation prevents a structured and rapid development of applications.

Elbfisch offers a robust runtime environment and a canon of easy-to-use basic functions that enable the team-based development of object-orientated automation applications in Java, when used in conjunction with common open-source tools such as:

- Netbeans/Eclipse – both highly-developed integrated development environments for the design of software applications incl. graphic interfaces (HMIs) and
- git,subversion, or similar – robust version management systems that are freely available on the market

Elbfisch combines well-established processes such as the cyclical processing of process signals with the modern concepts of object technology.

Elbfisch applications can be fully developed and tested on PCs using the operating systems:

- Windows XP, 7
- Mac OSX
- and Linux.

Porting to the target system, such as an embedded PC, is then only one small step away. The runtime environment, written in Java itself, implements the concepts:

- Module
- Event
- Signal

These concepts are described below.

Modules – directly corresponding to the real world

The Elbfisch design is based on the principle that an automation application should represent a copy of the process that it controls.

Industrial processes are generally made up of aggregates that may themselves be made up of subordinate aggregates.

Elbfisch's modular design supports the software-based representation of such hierarchies.

Consider the example of an industrial robot: the robot consists of a turning base, two arm joints, one hand joint and a claw. Each axis is fitted with a drive and a series of sensors that inform the axis of its position and the possibility of mechanical limits being reached.

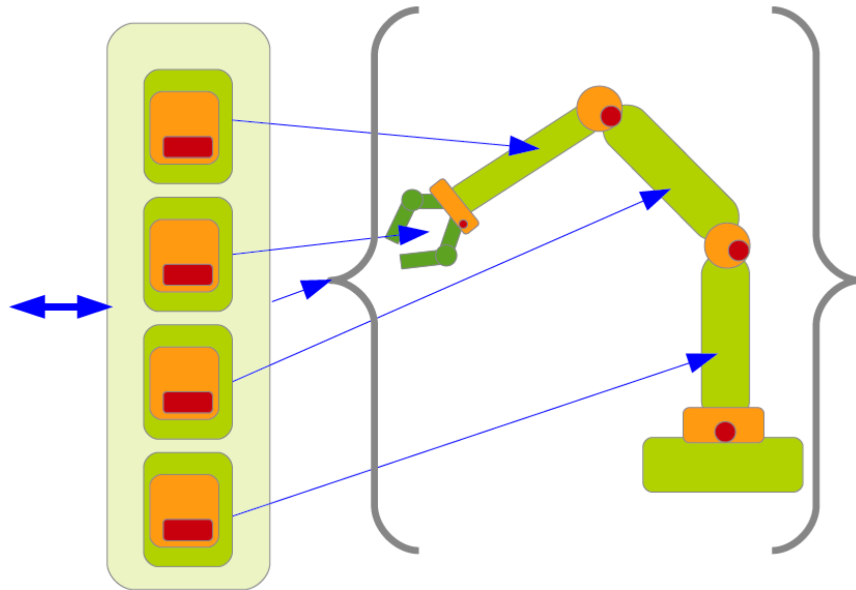

Fig. 1: Elbfisch – diagram of a process with modules

An Elbfisch application would be structured systematically, with each of these aggregates represented by a corresponding module. The application would implement the robot itself as an overarching module. The modules that represent the aggregates contain sub-modules, such as position controller and limit position monitoring, shown here in orange and red.

Each of these modules would be implemented without context. This means that each module is ideally unaware of the environment in which it is functioning. It simply makes its function available to adjacent or higher-level modules.

For example, the position controller could contain a higher-level PID controller that is implemented as a universal Elbfisch module and that comes from an external library.

Furthermore, our example clearly shows that certain modules are similar, or even identical.

This is the case for the turning base, for example. Once implemented, a second instance allows the module to be re-used for the hand's rotary axis.

Elbfisch provides a runtime system for the context-free and organised module interactions. This ensures that all modules involved in an application run in a real-time cycle.

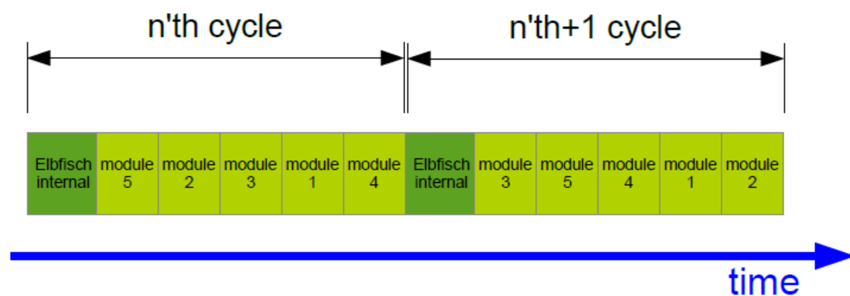

Fig. 2: Elbfisch – real-time cycle

The modules are started by Elbfisch as required and perform their task by processing their input signals and calculating output signals which they then send to other modules.

The sequence of the module workflow is not known and cannot be altered by the programmer of an Elbfisch application.

Signals – organised data exchange between modules

To allow the organised data exchange between modules that are implemented without context to be executed with no repercussions and consistently, Elbfisch uses the concept of signals.

Signals are formalised channels for data exchange between modules. Signals are created by modules and are by definition 'owned by' the module that created them. Only the owning module can write to the signal. All other modules can read the signal as required.

Synchronisation to the real-time cycle

The forwarding of signals is synchronised to the real-time cycle. This is an important mechanism that ensures that the signals exchanged between the modules remain consistent.

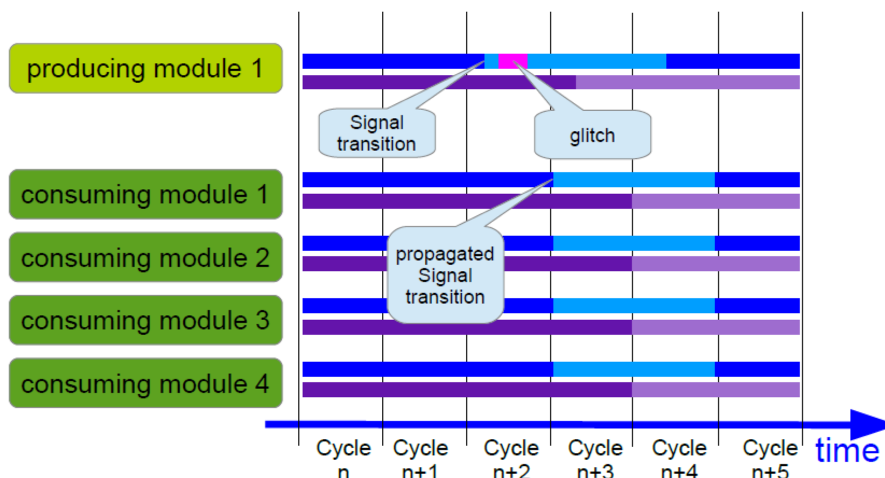

Fig. 3: Elbfisch – synchronised signal exchange

The above diagram shows a module that produces a blue and a violet signal (a producer). The module changes these signals at some point within the cycles n+2 and n+3 (depicted here by a colour change).

As the diagram shows, these changes are not visible to the four consuming modules until the next cycle begins. Even the temporary change to the blue signal (glitch), shown in magenta, does not disrupt the consuming modules. It is the stable value present at the end of the cycle that is forwarded to the other modules. Elbfisch performs this forwarding (propagation) at the start of each cycle (see also fig. 2). The signals in the application are consistent at all times. This is of great importance if modules implemented without context are to interact as described.

Using connections to link signals

Connections can be used to create multiple links between signals. If the original signal is changed, this change is propagated to all the reading signals in the cascade in the next cycle.

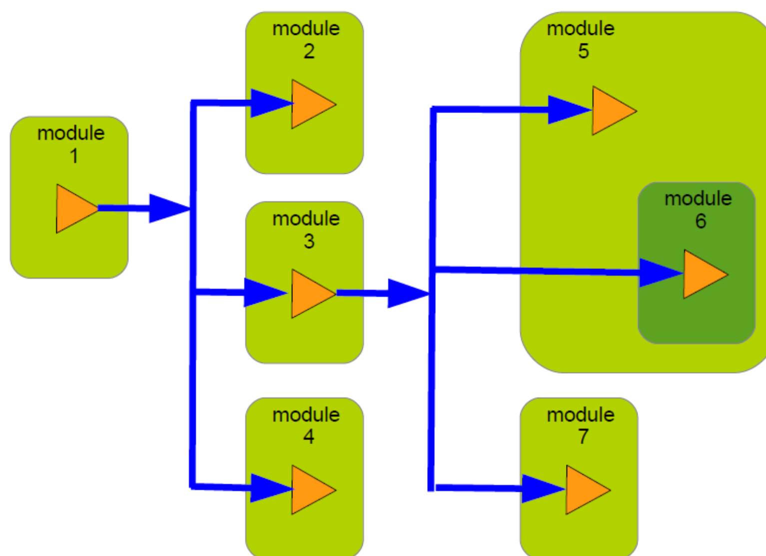

Fig. 4: Signal cascade

This creates the signal cascade shown in fig. 4, whereby a change in the original signal in module 1 is forwarded to all directly and indirectly connected signals when the next cycle begins. As a result, not only are modules 2, 3 and 4 informed of the change as soon as the next cycle begins, but also modules 5, 6 and 7.

The linked signals shown in fig. 4 cannot be written to directly, even by the owning module. The signals in the cascade can only be altered by the original signal.

Signals can be valid or invalid

In industrial processes, it is a regular occurrence that signals may become invalid. This may be the result of a bus system malfunction, a broken cable, a sensor defect or, quite commonly, an initialisation failure.

Elbfisch signals therefore normally have the characteristic of being valid or invalid. The signal is invalid by default before being set for the first time: it must be initialised for a certain purpose before it can be used by the subsequent modules.

If a module detects an internal error condition, it can set all of its output signals to 'invalid' to allow the consuming modules to react to the error status.

This status is propagated to all directly and indirectly connected signals in exactly the same way as a change in value.

Signal types – standard or user-defined

Elbfisch provides the following default signal types:

- Logical (boolean)
- Decimal (double)
- SignedInteger (int)
- CharString (String)

An unlimited number of signal types can be defined. Additional signal types can be implemented easily, provided they are suitable for the particular application – even structured signal types such as arrays, records, strings and objects, no matter how complex.

Events – representing the dynamics of a process

Events control the activation of modules. Modules are able to explicitly wait for an event to occur. When the event does occur, the module is activated in the next cycle where it performs the pending task and then returns to the status of 'waiting for specific event'.

- The range of possible events is unlimited. Examples include:
- The end of a set time period
- A decimal value being exceeded/not being reached
- A change in a digital input
- A signal becoming valid/invalid
- Any combination of events
- An error condition occurring
- An emergency stop condition occurring
- The application ending
- An event not taking place within a set time frame

In addition to the events described above, users can simply define their own events for specific projects and use them in their applications.

The programmer does not need to monitor the process for the occurrence of a certain event, as this is performed by the Elbfisch runtime system.

A module communicates its status of ‘waiting for specific event’ to the runtime system and then does not actively participate in the process until the event occurs.

The central work() method of an Elbfisch module is shown below:

```
©Override
protected void work() throws ProcessException{
   boolean done = false;
   InputSignalsValid inputSignalsValid = new InputSignalsValid();

   try {
      do{
         status.enter("processing");
         try{
            //wait, until all input Signal become valid
            inputSignalsValid.await();0
            done = false;
            do{
               try{
                  do{
                     double value = getAnaloglnput().get();
                     getAnalogOutput().set( -value);
                     getAnaloglnput().changes(value, 0.1).await();
                  }
                  while(true);
               }
               catch(SignalInvalidException ex){
                  //input signals have gone invalid
                  //invalidate own output signals
                  getAnalogOutput().invalidate();
                  done = true;
               }
            }
            while(!done);
         }
         catch(EmergencyStopException ex){
            Log.error("Error", ex);
            //invalidate own output signals
            analogOutput.invalidate();
            EmergencyStopAcknowledged esa = new EmergencyStopAcknowledged();
            esa.await();
         }
         status.leave();
      }
      while(true);
   }
   finally{
      //module will be shut down
      //invalidate own output signals
      getAnalogOutput().invalidate();
      status.resume(0);
   }
}
```

This simple example clearly shows how the event-controlled processing of signals in conjunction with Elbfisch works.

The basic example module simply inverts an analogue signal and then outputs it.

The module initially waits until all input signals are valid.

The corresponding event is called ‘inputSignalsValid’.

```
inputSignalsValid.await();
```

The module then changes its status to ‘processing’, where it reads an analogue input value in a loop and then outputs an inverted output signal.

```
double value = getAnalogInput().get();
getAnalogOutput().set( -value);
```

It then waits for the input signal to change by at least 0.1 of a given unit.

```
getAnalogInput().changes(value, 0.1).await();
```

The module is re-activated at this point as soon as the awaited event occurs.

The module performs the above task within the internal loop for an unlimited amount of time until it is interrupted, for example if the input signal becomes invalid. In this case, a SignalInvalidException is triggered. Our example module reacts by setting the status of its own output signal to ‘invalid’ to inform the consuming modules that it is currently malfunctioning.

```
catch(SignalInvalidException ex){
   //input signals have gone invalid
   //invalidate own output signals
   getAnalogOutput().invalidate();
   done = true;
}
```

The module then returns to the status where it waits for its own input signals to become valid.

Exceptions can be processed in catch() branches in Java, which supports programmers in programming the nominal sequence of their control tasks and treating exceptions separately. As a result, there is no need for all exceptions to be coded out at all affected program points.

This increases the program’s transparency, improving program quality.

Our example also shows how a module can take on various statuses and how, to a certain extent, these statuses can be directly depicted inline in code. Even complex status-dependent algorithms can therefore be implemented in a straightforward way.

A further example for the efficient programming of special conditions is the treatment of an emergency stop situation.

As soon as such a situation occurs, the Elbfisch runtime system automatically informs all modules via an EmergencyStopException.

```
catch(EmergencyStopException ex){
   Log.error("Error", ex);
   //invalidate own output signals
   analogOutput.invalidate();
   EmergencyStopAcknowledged esa = new EmergencyStopAcknowledged();
   esa.await();
}
```

Our example module processes this exception by first setting its own analogue output signal to invalid and then waiting for the emergency stop exception to be repealed. The third status of this example module, ‘waiting for repeal of emergency stop exception’, is shown below:

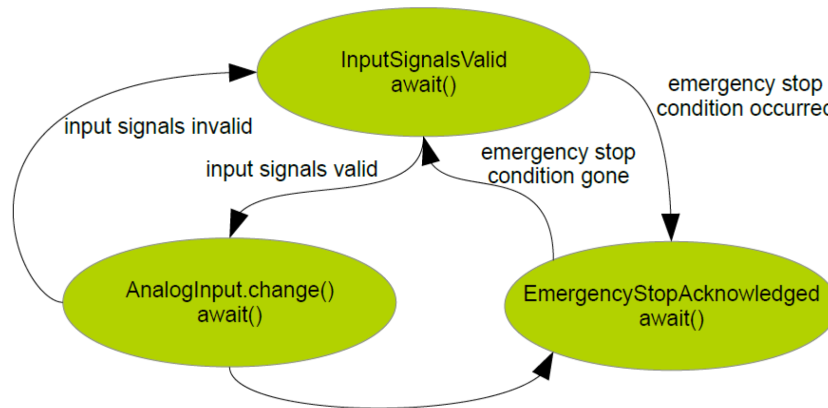

Fig. 5: Statuses and events in Elbfisch modules

The module then returns to the status where it waits for its own input signals to become valid – and so the process continues...

Elbfisch instances: distributed automation solutions with Elbfisch

Elbfisch provides a simple mechanism that allows multiple Elbfisch instances to cooperate via a conventional network. The modules are linked via ‘remote signals’. The signals are transmitted in frames that collect all the signals that are changed within a given cycle and transport them at one point in time.

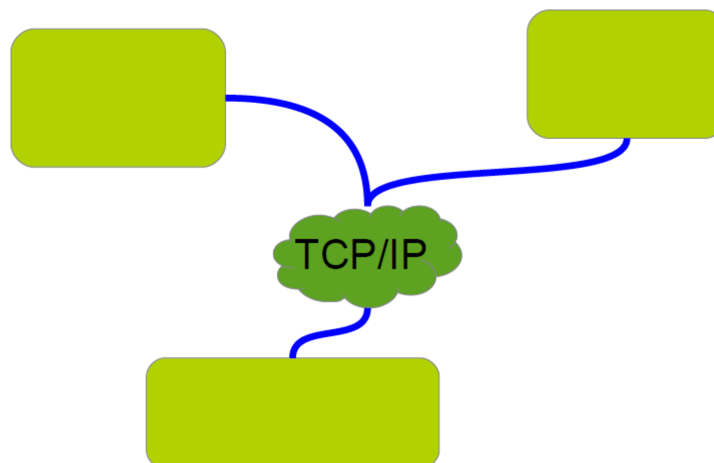

Fig. 6: Distributed Elbfisch instances cooperate via remote signals

The exchange of signals is based on the push principle: Whenever signals change within a cycle, these changes are collected in a frame and distributed to all Elbfisch instances for which these signals are relevant.

Hierarchical configuration

All automation applications need parameters and configuration data. Elbfisch keeps this information in a central XML-based configuration file.

Each module can easily store the configuration data for its specific class or instance in this file.

If we think back to the example of the robot, all the axes need to be parametrised individually, even if they are fitted with identical controllers. At the least, the process signals will have different I/O signals.

A section is automatically entered in the configuration file for each instance of a module, allowing the instances to be parametrised individually.

The Elbfisch project

The runtime system is stable and so complete that it can be used to realise relevant automation applications.

The projects realised with Elbfisch by now have shown that Elbfisch can be used efficiently for a range of applications.

Drivers are available for communication with S7-CPUs via ISO over TCP and with modbus slaves (e.g. Wago terminal blocks).

The Elbfisch project is nevertheless, like all other open-source projects, sustained by the contributions made by others. The roadmap covers the implementation of additional drivers for established industrial buses such as EtherCAT and ProfiNet, the implementation of a graphic user interface as a plug-in for IDEs such as NetBeans or Eclipse and a more efficient design for Elbfisch applications.

We do not want to restrict the dynamics of the development in any way.

Please feel free to take a look at www.elbfisch.org. Additional examples of code and binaries (jars) and sources to download can be found there.

Bernd Schuster

MSK Gesellschaft für Automatisierung mbH

www.mskgmbh.com

www.elbfisch.org