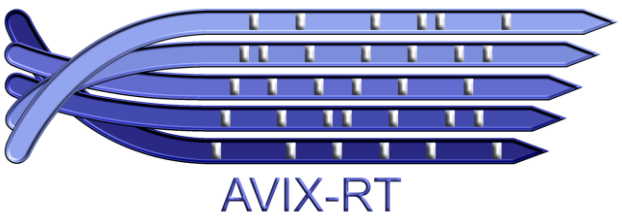
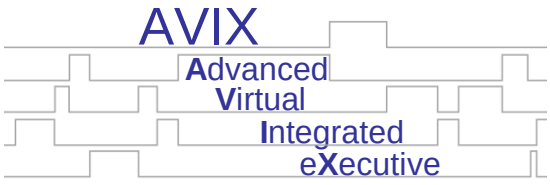
AVIX Real Time Operating System

The AVIX Exchange Mechanism

An Example Application





AVIX-RT
Maïsveld 84
5236 VC 's-Hertogenbosch
The Netherlands

phone +31(0)6 15 28 51 77
e-mail info@avix-rt.com
www www.avix-rt.com

Table of contents

Table of figures

1 Introduction

It can be quite challenging to create a full blown application. Testing and integrating individual software components may require a lot of work. Every component potentially uses different interface mechanisms, tightly coupling it to other components. This makes testing challenging because test code has to be able to work with every interface mechanism used. Adding new functionality not only requires new code to be added but often many changes to existing code must be made. Combining software components running with different periods may require custom solutions to prevent a waste of valuable system resources. Reusing software components in another application may require many changes to this component and thus a complete retest of that component.

AVIX Exchange Services offer a solution to all these challenges. Using Exchange Services, all interfaces between software components are based on a single, uniform technology, AVIX Exchange Services. A thread producing data just writes to an Exchange Object and is done. A thread consuming data just connects to an Exchange object and is notified when new data is written by a producer. Individual threads have no knowledge of each other and only use Exchange Objects for writing and reading data.

Exchange Services are based on the proven publish-subscribe design pattern. Every individual software component is highly modular, modularity bringing many advantages. Exchange Services are fully integrated in AVIX and enhances its functionality in a way not offered by any competing product.

This document presents an Exchange based application containing four software components together offering the desired functionality. All steps, from the functional requirements up to the final implementation are presented to provide detailed insight in the advantages offered by Exchange services when it comes to application construction.

The AVIX-RT website offers Free AVIX Demo distributions[1] for a large number of supported microcontrollers and development boards. For those development boards capable of running the application presented in this document, the demo application presented here is offered as working code allowing even more insight in the power of Exchange services and offering a way to experiment on real hardware.

The application presented in this document is called the Extended Demo. Besides providing detailed insight in the AVIX Exchange mechanism, the Extended Demo uses many of the other mechanisms offered by AVIX like threads, pipes, messages and event flags. The reader should have basic understanding of these mechanisms to benefit the most from the information presented here.

Three sections are presented in this document.

Chapter 2; Extended demo application presents the mentioned application in sufficient detail to see the advantages of AVIX Exchange Services

Chapter 3; Testing an AVIX Exchange based application presents an approach how AVIX Exchange Services can assist in testing application components.

Chapter 4; Extending an AVIX Exchange based application illustrates how AVIX Exchange Services may be beneficial when new functionality needs to be added to an existing application.

[1] Free AVIX Demo Distributions are for evaluation purposes only. Be sure to read and accept the License Agreement before downloading and using such Distributions.

2 Extended demo application

The demo application presented in this chapter is a fully functional application using many of the AVIX supplied services, the most important being Exchange services. The approach to present the application is top-down, first specifying the requirements, next creating a global design and finally a detailed design of the individual software components the application is composed of.

2.1 Functionality and Context

Decided is to create an application with the following functionality:

1. Two analog inputs (channels) are sampled with a user adjustable sample period from 2ms to 10ms.

2. The analog values and the current sample period are shown on an LCD

3. The analog values are send to a serial port (UART) in ASCII format

4. The user can make a selection of the channels shown on the LCD and send to the serial port. For this purpose the application can be controlled from the development board using available switches of by sending a command character to the application using the serial input.

5. The user can select the desired sample rate between 1ms and 5ms in 1ms steps. For this purpose the application can be controlled from the development board using available switches or by sending command characters to the application using the serial input.

Figure 1 shows the application, its peripherals and the interaction with the user.

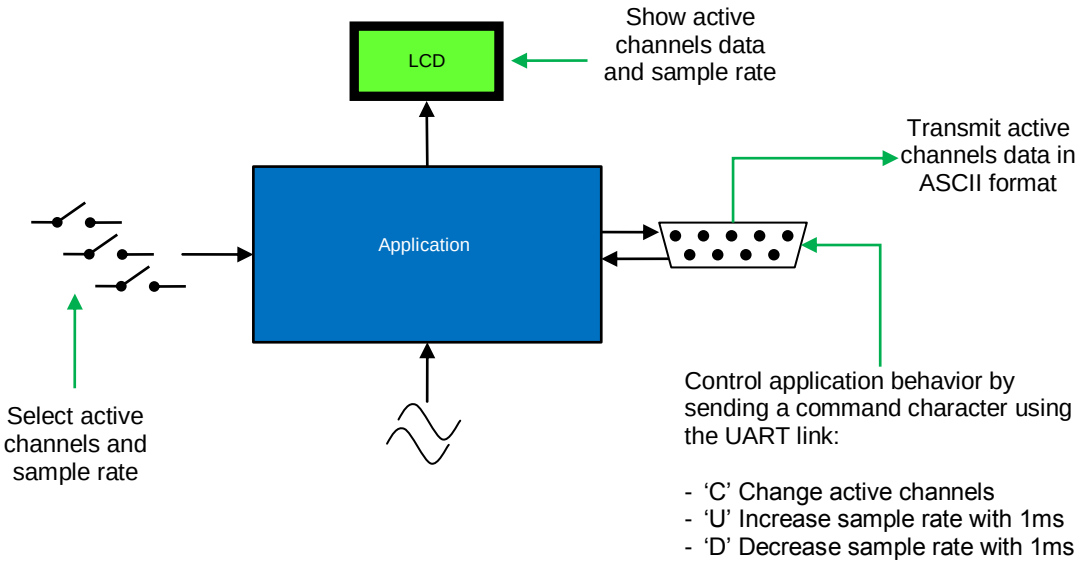


Figure 1: Application Context and Interfaces

2.2 Top Level Design

Based on the requirements, four top level software components are identified:

1. Communication Handler Component
2. LCD Handler Component
3. Switch Handler Component
4. Analog Handler Component

One of the challenges is how to 'tie' these software components together. Often a selection is made from the mechanisms offered by the underlying RTOS like pipes, messages and so on. Although usable for this purpose, they do have a drawback in that they hurt application modularity.

AVIX offers a service which is specifically intended to be used at this level of application development, Exchange Services. Exchange Services offer Exchange Objects. Exchange Objects can be considered thread safe global variables with a built in notification mechanism. When an Exchange Object is written, the notification mechanism informs connected software components of the fact that new information is available. Multiple software components can be connected to an Exchange Object so writing to an Exchange Object can inform many interested subscribers.

Very important is that a Software Component writing to an Exchange Object does not know which, if any, Software Components are connected to the Exchange Object. Likewise, a Software Component connected to an Exchange Object does not know which Software Component is writing the Exchange Object or which other Software Components are connected.

The use of Exchange Objects leads to a highly modular application design allowing individual Software Components to be developed and tested in isolation.

For the application developed here two Exchange Objects are required. One holds the application control information with the active channels and the sample rate (Control Exchange), the other is used to hold the analog values from the ADC sampling process (Analog Exchange).

Figure 2 shows the top level Software Components and the Exchange Objects tying them together.

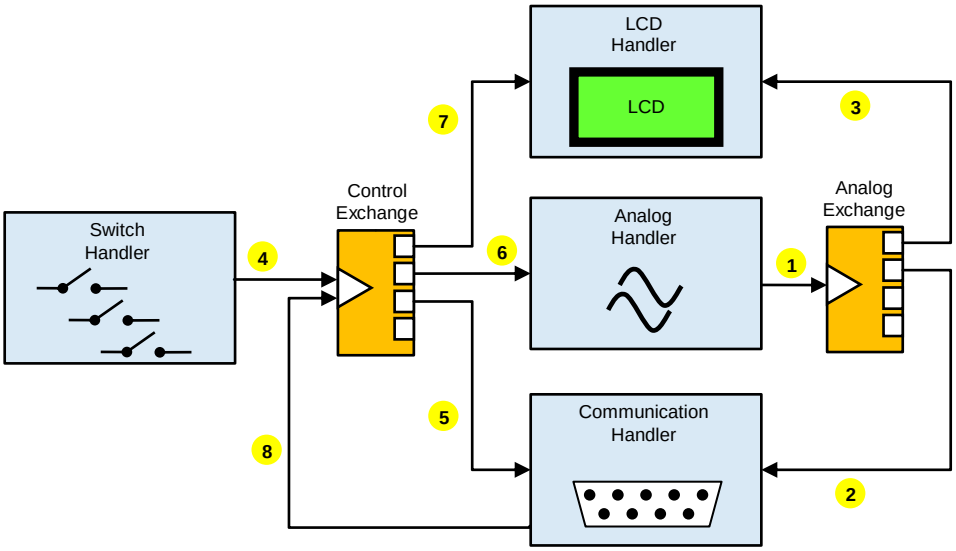


Figure 2: Top Level Components and their interfaces

In this figure, an Exchange Object ingoing arrow denotes a write action. An outgoing arrow denotes a so called connection informing the connected Software Component of the fact that new information is available.

In Figure 2 the following information flows can be recognized:

When new analog data is generated by the Analog Handler this data is written to the Analog Exchange (1). As a result, a notification is send to the Communication Handler (2) and the LCD Handler (3). The Communication Handler needs the analog data to serialize and send, the LCD Handler needs the analog data to show on the LCD.

When a switch is pressed on the development board, the Switch Handler interprets this and decides what to change in the Control Exchange (active channels or sample period). The Switch Handler writes the new information to the Control Exchange (4). All other Software Components are notified of this change so they can act accordingly. Through (5), the Communication Handler is informed so it can read the active channel state in order to know which analog channels to send. Through (6), the Analog Handler is informed so it can change the sample period according to the new value. Finally through (7), the LCD Handler is informed. This Handler uses the active channel setting to decide how to display analog values and the sample rate to display the selected value.

Very interesting is flow (8). As described in the functional requirements, the application can be controlled by switches and through commands received on the serial link. When commands are received on the serial link, just like a switch being pressed on the development board, this just results in an update of the control information in the Control Exchange. This update leads to connection (4), (5) and (6) being notified just like before.

As shown, even the ingoing and outgoing data of the Communication Handler are isolated from each other. When a command is received on the serial link, the fact this will result in a change of the serialized channels is not coded inside the Communication Handler itself. This structure has two advantages; first the design of the Communication Handler is ‘cleaner’ and more straightforward; second, it prevents the Communication Handler from containing code to inform other components which does not belong there. All the Communication Handler ‘knows’ is the Exchange object and what other components are connected to this is of no interest of this component.

At this stage of designing an AVIX based application, a very interesting milestone is reached. Because of the use of Exchange Objects the software components are entirely decoupled from each other. Known are the required software components and their interfaces. These interfaces are to Exchange objects only and there is no dependency between the individual software components. As a result, each of the software components can be individually detailed further without taking into account constraints placed upon them by other software components.

2.3 Detailed Design

From this point on, every individual Software Component can be designed in more detail in isolation. Design decisions can be taken for every individual Software Component without taking into account what is going on in other Software Components and without worrying about the effect of local design decisions made for other Software Components[2]. It is in this step that other AVIX Services like pipes, event groups and messages are used to obtain the desired behavior.

Communication Handler

Let's illustrate this in detail for the Communication Handler and see how the detailed design of this Software Component will look. To start, let's isolate the Communication Handler from the total application taking into account its Exchange Object based interfaces. This is shown in Figure 3..

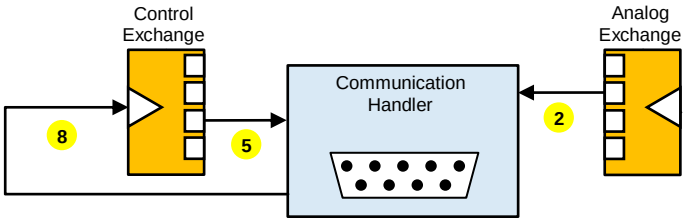


Figure 3: Communication Handler Components and its interfaces

To design the Communication Handler, a number of design decisions are made. These are dealt with next. The result of these decisions is shown in Figure 4 and elaborated on next.

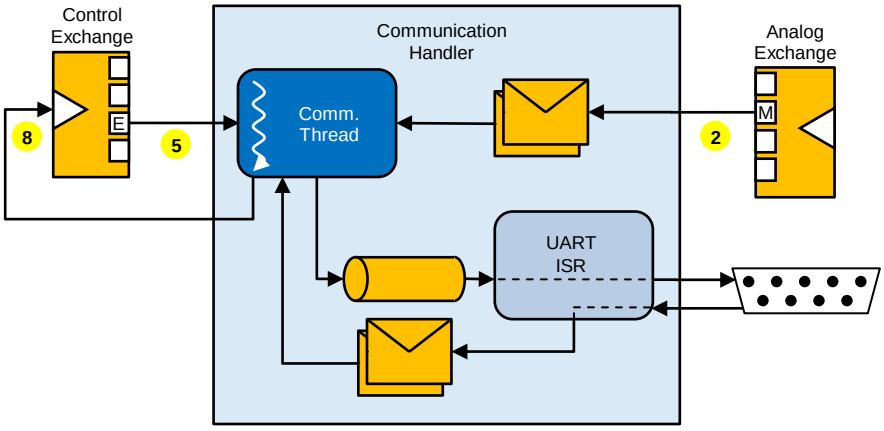


Figure 4: Communication Handler detailed design

To decouple the processing of the Communication Handler from the rest of the application, a thread is used for executing the majority of the Communication Handlers functionality. Since device handling is preferably done using interrupts to decrease system load, an ISR is used to deal with the low level aspects of the serial communication.

[2] It shall be obvious a relation between the different Software Components *does* exist. Think for instance of timing aspects where one component will potentially influence another. Of course this has to be taken into account when designing the total application.

The Communication Handler must be connected to the two Exchange Objects. When new data is written to the Analog Exchange, the Communication Handler must be informed of every individual value so this must be a *push connection*[3]. Chosen is to use a message based connection so every value written to the Analog Exchange results in a message being send to the connected thread.

When new data is written to the Control Exchange, on the other hand, the Communication Handler does not need to react immediately. Only when new data is send out, the Communication Handler must do so according the status of the Control Exchange. So it is sufficient for the Communication Handler to read the Control Exchange at that moment. Still it is advantageous to know that the Control Exchange *does* contain new data so no redundant reads will be performed. So here a *pull connection*[3] is chosen in the form of an event group connection. When new data is written to the control Exchange object, an event flag is set to inform the Communication Handler.

When the thread has new data available to send to the serial link, a Pipe is very suitable. The thread can just write to the pipe and using the pipe callback mechanism the UART ISR will be informed new data is available to transmit. This callback can then activate the transmit interrupt and the ISR will read the data to be transmitted from the pipe.

Finally when a command is received on the serial link, the ISR will allocate a message, place the received command character in this message and send the message to the thread. The thread can then interpret the command and write the Control Exchange accordingly to inform the rest of the application of the received command.

As shown, all decisions regarding the Communication Handler can be taken without having any influence on the rest of the application. Exchange objects offer an interface mechanism leading to a very high level of modularity. When needing the Communication Handler services, all that needs to be done is write to the applicable Exchange object.

Likewise the other three software components can be designed in isolation from the other components since they all interface with Exchange objects only.

LCD Handler

The responsibility of the LCD Handler is to present information to the user. This typically is a slow process compared to the rest of the application which is very important to consider. In this example, new ADC data will be provided once every millisecond. Not only will it probably be impossible for the LCD to keep up with this speed, it is not required since the human eye is too slow to see every individual update at this rate.

Based on this, the first decision is for the LCD Handler to have a pull connection to the Analog Exchange. Every write operation to the Analog Exchange results in an event flag being set in an LCD handler local event group. Added to the LCD handler is a timer that expires every 50ms. This time is chosen to obtain an LCD update rate of 20 times per second.

Now only when both the timer expires *and* the mentioned event flag is set, the LCD thread wakes up. At this moment the content of the Analog Exchange is read and written to the LCD. In this situation a lot of intermediate writes to the Analog Exchange get lost as far as the LCD Handler is concerned but this does not matter in this case.

For the same reason as mentioned with the Communication Handler, for the Control Exchange a pull connection is chosen. The resulting detailed design of the LCD handler is shown in Figure 5.

[3] Exchange objects offer many different types of connections. A connection belongs to the *push* or the *pull* category. The essence of push connections is that no data will get lost. Every write to an Exchange object results in the new data being transferred to the subscriber. Pull connections on the other hand inform the subscriber of the availability of new data but it is up to the subscriber to read this data at a convenient moment. Subsequent writes may overwrite the data that originally triggered the connection so data of intermediate writes may get lost. Although this may sound strange, pull connections are especially useful for components intended for user interaction. For more details, see the AVIX User and Reference Guide.

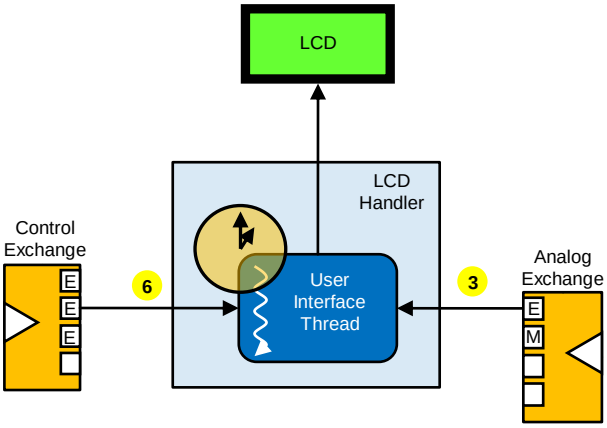


Figure 5: LCD Handler detailed design

Switch Handler

The responsibility of the Switch Handler is to deal with the switches and translate individual switch presses to the desired action. For reason of efficiency again, at the lowest level, use is made of interrupts to be informed of a switch being pressed.

Once a switch is pressed, a thread is informed of this. Reason is the switch needs to be de-bounced, a delay of a few milliseconds is required before deciding the switch is actually pressed or not. This is the responsibility of the Switch Handler thread since this has access to AVIX timers for this purpose.

Once this thread decides a switch actually is pressed, it translates the physical switch to the desired new content of the Control Exchange and writes this Exchange accordingly so other Software Components are informed of the desired action. The resulting detailed design of the Switch Handler is shown in Figure 6.

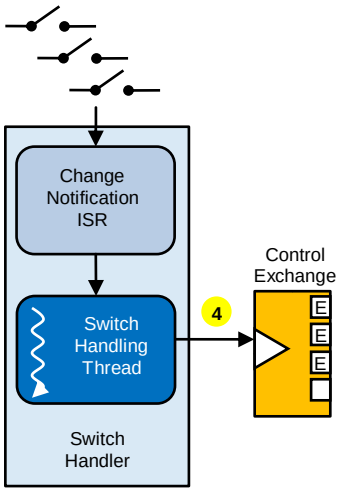


Figure 6: Switch Handler detailed design

Analog Handler

The Analog Handler finally forms the core component of the application. This handler is responsible for generating the data the application is intended for in the first place. Based on the requirement this data should be generated with a fixed period between 1 and 5ms, a thread is chosen again with an AVIX timer waking this thread up when it expires. This timer is set to a period based on the content of the Control Exchange. Since this period may change, a pull connection is created to the Control Exchange.

Only when the timer expires, the flag set by this pull connection is tested to see if the timer period must change. A valid question would be why not to use a push connection and change the timer period the moment a new value is written to the Control Exchange. The problem is that it cannot be predicted at what moment during the current period the Control Exchange will be written. Suppose the current period is 5ms and when 3ms of this period have elapsed, the Control Exchange is written with a new period value of 4ms.

Would the timer be reconfigured at that moment, effectively one period of 7ms would be the result, namely the 3ms already elapsed and the 4ms of the new timer period. This is not desirable. So when a new period is written to the Control Exchange, the current timer period is finished before reconfiguring the timer with the new period. Doing so, the 5ms periods are followed by 4ms without the mentioned effect and for this reason a pull connection is the right choice here.

When the Analog Handler is activated because its timer expires it has to obtain the ADC data. Again, low level device handling is done using an interrupt handler. By using an AVIX pipe between the thread and the ISR, all the thread needs to do is read from the pipe. This pipe is empty and the thread will block. Connected to the pipe is a callback function which will trigger the ADC device. Once the conversion is ready, the ADC device will generate an interrupt. The ISR will run and write the analog data to the pipe. This will wake the thread which deals with this data further. In this example all that is done is write the data to the Analog Exchange for other components to pick it up. In a real application it is likely some processing will be done before passing the data on. The resulting detailed design of the Analog Handler is shown in Figure 7.

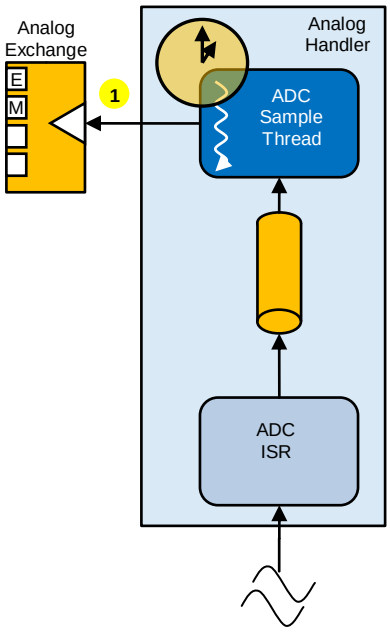


Figure 7: Analog Handler detailed design

The resulting application

The total detailed design with the content of every software component based on the discussion above is shown in Figure 8.

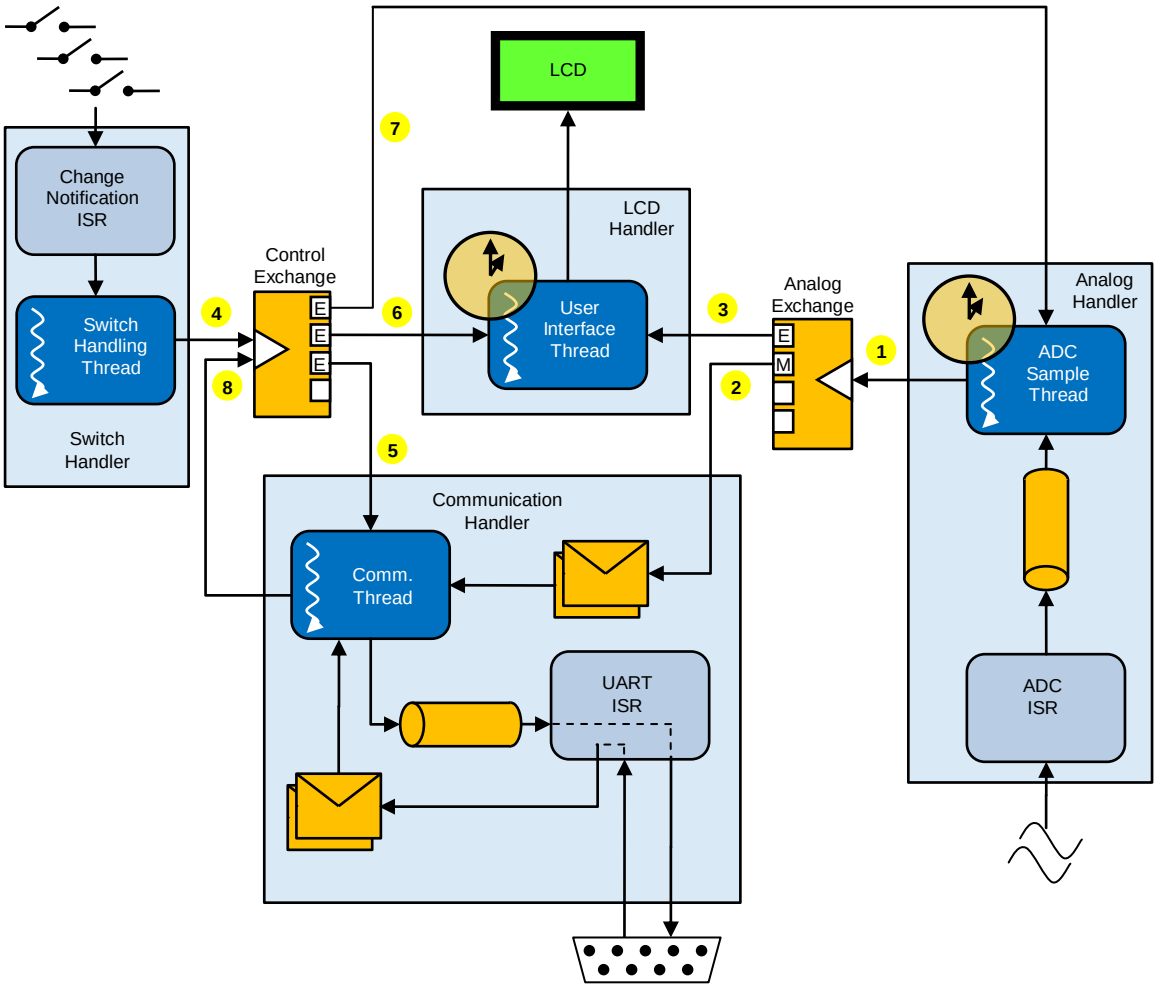


Figure 8: Total Application Detailed Design

3 Testing an AVIX Exchange based application

AVIX Exchange Services are especially useful when testing an application, parts of the application or individual software components. Also monitoring data generated by individual software components can be added for test purposes without the application being changed.

When using Exchange Services as the interface mechanism between software components, this is the only technology test code has to use. As a result, test code becomes more uniform and less error prone.

3.1 Testing individual Software Components

When testing a software component, the Exchange based interfaces of that component are used to provide test data and to obtain the result of the processing performed by the component.

Not a single change to the code of the software component under test is required. When a software component produces data, this data is written to an Exchange object. The component does not know who is connected to the Exchange object, whether this are one or more other software components belonging to the application or, in case of testing, a test component. As a result, the code that is tested is equal to the code being used for the resulting application.

Figure 9 shows an example how the Communication Handler is isolated from its application and its interfaces consisting of the Exchange objects being written by a Test Handler.

Note once more that the Communication Handler does not require a single change for this and when running it does not know that new data is provided by another component in this case.

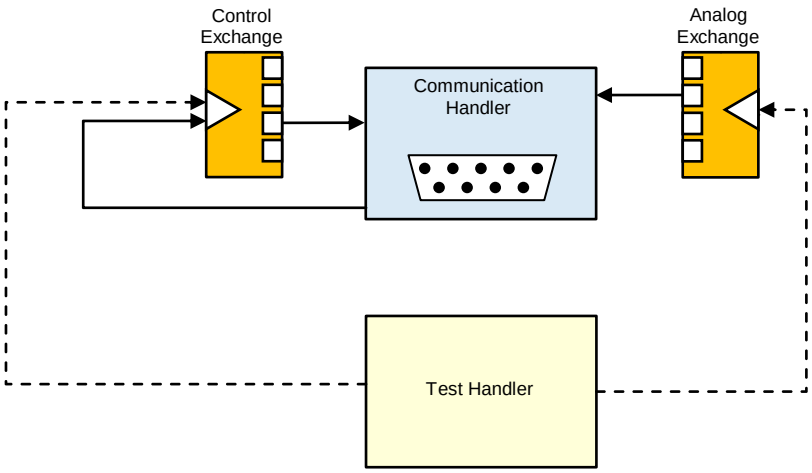


Figure 9: Component testing using Exchange Services[4]

[4] This example happens to be with two Exchange objects written by the Test Handler. It shall be obvious that for software components writing to Exchange objects, a Test Handler is equally capable of connecting to such an Exchange Object. In such a case the Test Handler can both provide data required by the software component and observe the result it produces.

3.2 Testing larger parts of an application

Exchange Services also allow easy testing of larger parts of the application. It is easy to temporarily replace a component with a test component to provide sequences of test data. An example of this is shown in Figure 10 where the Analog Handler is replaced by a Test Handler.

In this example, the Test Handler can for instance be used to generate predefined sequences of data and test whether the reaction of the rest of the application is as expected.

Again, none of the other components need any change for this to work.

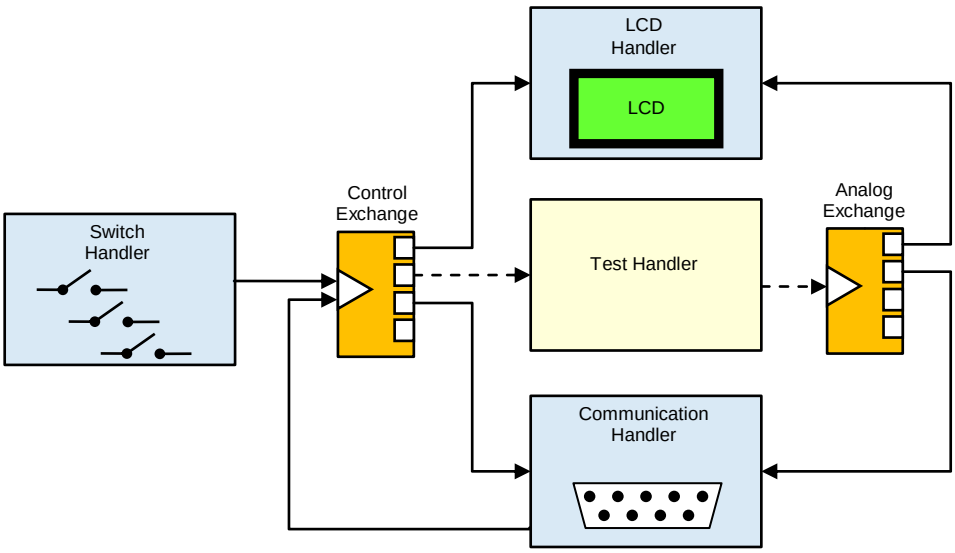


Figure 10: Application testing using Exchange Services

3.3 Monitoring software components

Exchange objects allow data produced by individual software components to be monitored for logging or validation purposes. When using Exchange Services, all that needs to be done is add a monitoring component by letting it connect to the desired Exchange object(s). Again, not a single change to other software components is required and they are not aware s monitoring component is intercepting the data they produce.

Figure 11 shows the demo application presented before where a Monitor Handler is added. This handler creates a connection to the available Exchange objects and when the application is running, this Handler will be informed of every write operation to these Exchange objects. At the risk of repeating, not a single change of code to the application components is required.

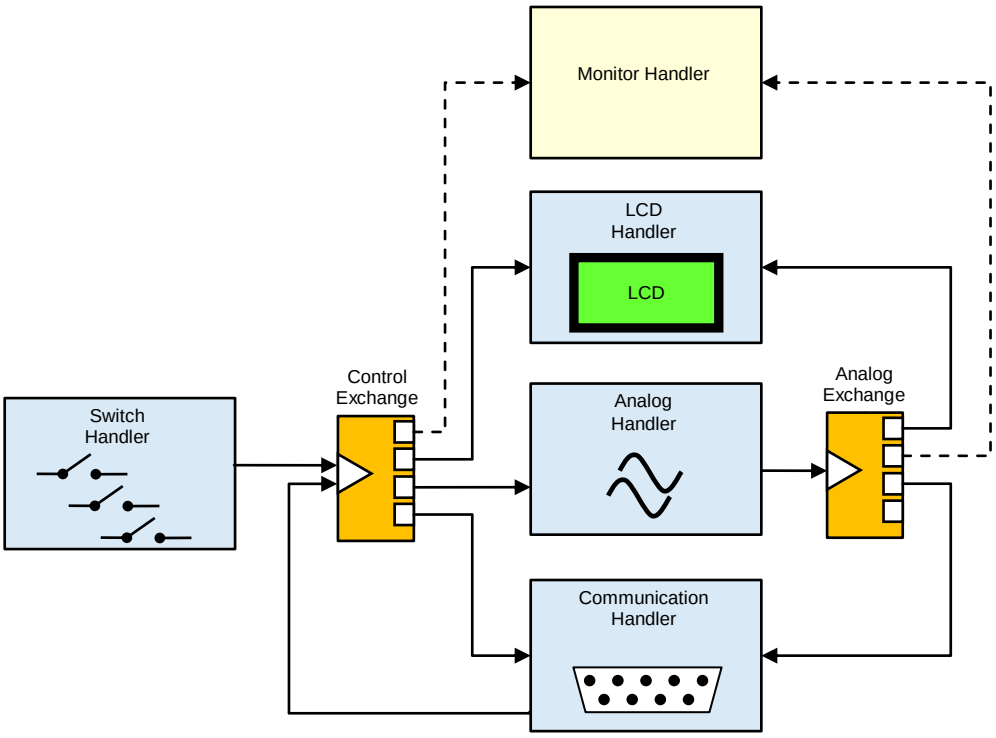


Figure 11: Monitoring an application for test purposes

4 Extending an AVIX Exchange based application

AVIX Exchange Services allow for easy upgrading of an application by adding software components that just need to interface to existing Exchange Objects. This is illustrated by an example. Suppose the application presented in this document needs to be extended with a network interface allowing the application to be controlled from a PC connected to the network.

Actually, this can be considered just an additional user interface. The PC application will show the analog values produced by the application and buttons on the PC screen allow the application to be controlled.

To accomplish this, a software component is added being responsible for communicating to the network; let's call this a network handler. The result is shown in Figure 12.

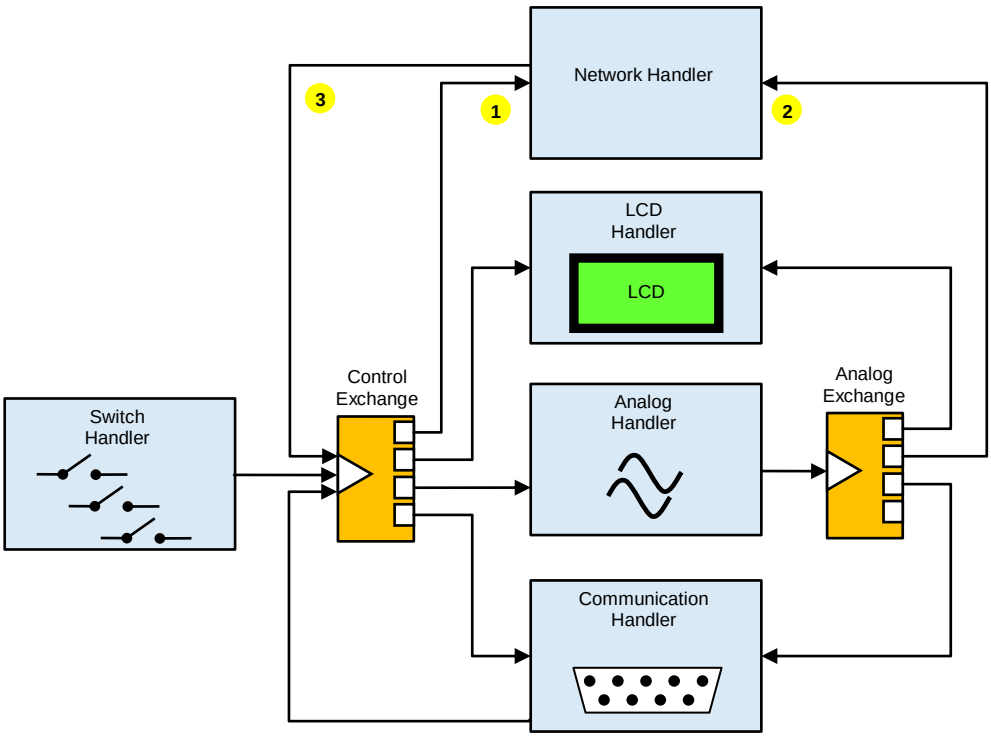


Figure 12: Extending an Exchange based application

All that is required is to add the new handler and connect it to the existing Exchange object.

Connection (1) informs this handler a new sampling period is selected or the active channels have changed. The handler will convert this information to a HTTP message and send it to the PC.

Connection (2) presents the analog data. The Network Handler will treat this the same as the LCD Handler does and when a new update must be generated, again, the data is converted to a HTTP message and send to the PC.

When the Network Handler receives a command over the network, originating from the PC application, it will translate this to a write operation to the Control Exchange (3) and the application will react accordingly.

Do realize the advantages of this approach. Not a single component has any knowledge of the other components present in the application. Still everything works as expected.

When the user changes the sampling period by pressing a switch on the development board, the LCD handler will present the new period on the LCD, the Network Handler will send the new period to the PC application and the Analog Handler will change its sampling period.

The exact same happens when the sampling period is changed because a button is pressed in the window of the PC application.

The result of using Exchange objects as interfaces between individual software components is a very high level of modularity where the advantages are obvious.