

# **Experiments: Labview and RS232**

**September 2013**

**Dušan Ponikvar**

**Faculty of Mathematics and Physics  
Jadranska 19, Ljubljana, Slovenia**

There are many standards describing the connection between a PC and a microcontroller to exchange data (USB, RS232, GPIB...). The RS232 standard describes wires, connectors, and signals to make a simple and reliable data transfer. The hardware implementing this standard is available in most of contemporary microcontrollers. This text describes the programming needed on the side of the PC running Labview to connect to the microcontroller using RS232 and exchange data.

The experiments will be performed on a commercially available demo board STM32F4Discovery utilizing the microcontroller STM32F407VG. The complete data on the board is available at [www.st.com/stm32f4-discovery](http://www.st.com/stm32f4-discovery). The board will be programmed during the course by a program which is available at the lecturer's home page. The software converts the board into a simple multifunction PC interface.

### Hardware requirements

The following hardware will be used:

- board STM32F4Discovery with USB-A to mini USB-B cable to supply the board,
- PC computer,
- USB to TTL-RS232 converter cable, and
- connectors, resistors, capacitors and wires to connect the above and also to connect signals to the Discovery board.

An oscilloscope, function generator and similar electronic instruments will be used.

The RS232 signals "Transmit data" (TXD) and "Receive data" (RXD) come in two flavors:

- The most widely available RS232 signals at the D9/Serial connector of a standard PC, and jump between about -12V (logic high) and about +12V (logic low). The voltage levels for logic high and low may differ from the above, anything between +5V and +15V is considered logic low, and anything between -5V to -15V is considered logic high.
- The ones used at pins of a microcontroller jump between +5V for logic high and 0V for logic low. Such signals are nick-named "TTL-RS232". Here also the voltage levels may vary depending on the logic family used and the power supply voltage.

In order to connect signals TXD and RXD from a microcontroller to signals RXD and TXD on a PC a conversion between voltage levels is mandatory, and the unit for conversion must also invert the signals. The signal TXD from one device (here the transmitting device shouts out the data) should be connected to the other device, input RXD (here the receiving device listens to data).

An integrated circuit like MAX232 (Maxim) can be used to convert the voltage levels and invert signals. The chip is placed next to the microcontroller; the principal diagram is given in Fig. 1.

Most modern PCs and laptops are equipped by USB ports, and some do not even have RS232 D9 connectors. Signals at USB port are completely different from RS232 signals, but can be converted to RS232 signals by an interface based on chip FT232 (FTDI). The diagram of such connection is given in Fig. 2. We will use this option for our experiments.

The Discovery board has pins for RS232 signals, these are:

- TXD (transmit data) is located at connector P1, lower quarter, pin PD8,
- RXD (receive data) is located at connector P1, lower quarter, pin PD9, and
- GND (ground) is conveniently located at the bottom of connector P1.

Note that connecting signal TXD from microcontroller with signal TXD from the FTDI chip may cause a failure. Make sure that TXD from one device is connected to RXD of the other!

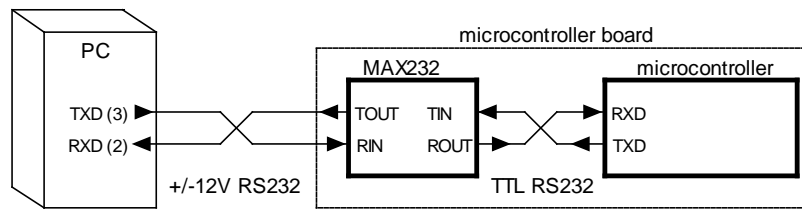


Figure 1: A connection between a PC and a microcontroller using the RS232

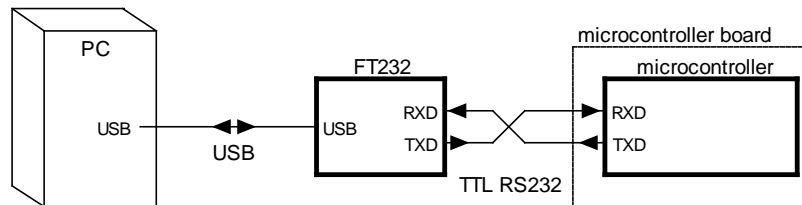


Figure 2: A connection between a PC and a microcontroller using mixed RS232 and USB

The pins to connect signals for experimenting are available at connector P1 on the discovery board as shown in Fig. 3. Please note the addition of resistors and a capacitor. These are added to ease the measurements and simplify the connection of oscilloscope probes to the resistor wires. They are not intended to protect the Discovery board against an overvoltage! Please take care about static discharge and make sure that the voltages connected to the pins of the Discovery board remain within the power supply rails of the microcontroller between 0V and 3V.

## The Discovery board – software

The Discovery board running this software can perform different functions. Each function is invoked by a command from the PC, and most functions take arguments. Each command starts with a letter '\$', and is terminated by character CR (carriage return, 0x0d). A typical command might look like:

```
$C100,200,0(CR)
```

where '\$' is the character to signal the start of the command, character 'C' is the code for the function to be performed at the Discovery board, 100, 200 and 0 are comma separated arguments for the function, and (CR) is the carriage return character. There may be up to five arguments for certain functions in the current version of the software.

Such command can be issued from any terminal program by simply typing it in and pressing the return key on the keyboard. Spaces are ignored, and lower case characters are simply echoed back to the PC. Error checking on the side of the Discovery board is not implemented, so if one makes an error in typing, the Discovery board will not be able to perform.

Some functions return results. These may be strings of characters or strings of bytes, and the length of strings is not necessarily known in advance. The return strings can easily be observed using a terminal program that can interpret received bytes either as characters or as hex values.

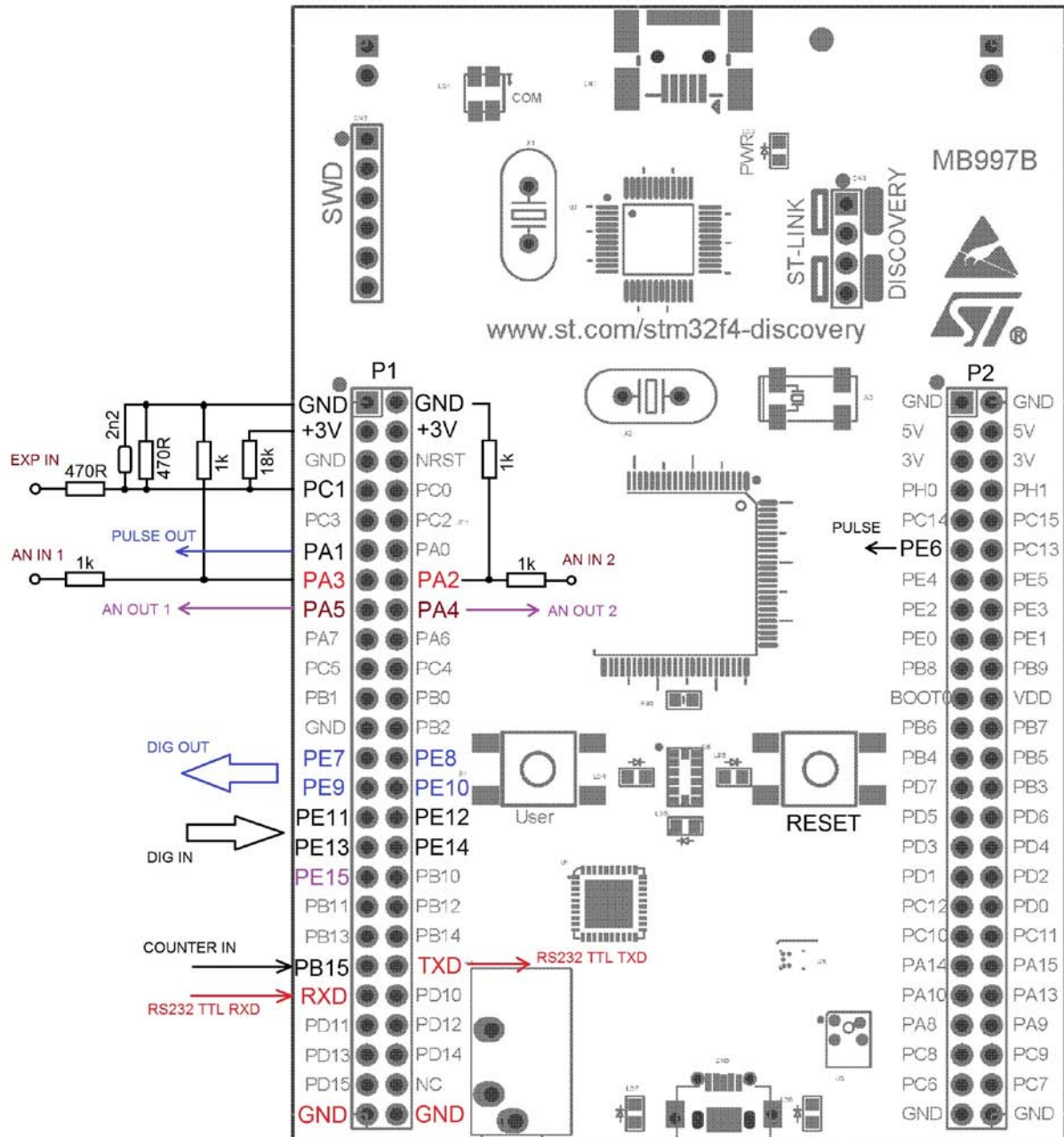


Figure 3: The layout of test signals at the Discovery board

The Discovery board implements high **baud rate of 921600 bps**. Such high rate is selected to allow fast data transfer during continuous sampling of analog voltages.

### List of functions

#### I. Introduce itself

This function is an exception to the rule stated above and is invoked by sending a single character '?' to the Discovery board. The response to this command is a string of characters, currently "Discovery – Demo: V1.1". There are 23 characters to be sent, and this takes about 230µs. In principle these characters should be available in the buffer of the PC within less than a millisecond taking into account all possible delays and latencies.

II. Input digital data

This function is invoked by command “**\$I(CR)**”. Once running it reads digital values from pins PE11, PE12, PE13 and PE14, and sends them to PC in lower four bits of a single byte. The returned byte should be available within less than a millisecond.

III. Output digital data

This function is invoked by command “**\$H10(CR)**”, where 10 represents an argument. The argument above is interpreted by the function as binary number 1010, and used to set pins PE10, PE9, PE8 and PE7 to values 1, 0, 1, 0.

IV. Output static voltage on two DAC channels

This function is invoked by command “**\$E1000, 2000(CR)**”, where 1000 and 2000 represent two arguments to set output voltages at pins PA4 and PA5 respectively. Please note that the full range of the DAC is limited by 0 and 4095, and this are also the limits of the allowed arguments.

V. Output two sine wave signals having adjustable frequency, amplitude and phase

This function is invoked by command “**\$C5,200,1024(CR)**”, where 5 represents the frequency, 200 represents the amplitude, and 1024 represents the phase angle between the sine waves available at outputs PA4 and PA5. Please note that the argument for frequency is not calibrated, but bigger number makes bigger frequency. Also, the allowed argument for the amplitude is between 0 and 250. The argument for the phase is limited to 4095 causing the phase angle of 360 degrees (1024 makes 90 degrees). The generation of sine waves can be stopped by command “**\$D(CR)**”.

VI. Output digital pulses having adjustable period and width

This function is invoked by command “**\$F84000, 5000(CR)**”, where 84000 stands for the number of clock ticks in a period (84000 == 1ms), and 5000 stands for the width of the pulse (5000 == 59.5µs). The train of pulses is available at pin PA1. The generation of pulses can be stopped by command “**\$G(CR)**”.

VII. Take a sample of two analog voltages simultaneously

This function is invoked by command “**\$A(CR)**”. The Discovery board will take samples of input voltages at pins PA2 and PA3, and return four bytes to PC computer within a millisecond. Bytes 0 and 1 carry high and low bytes of the result for pin PA2, and bytes 2 and 3 carry high and low bytes of the result for pin PA3. The software on the PC should combine low and high bytes of the result into a common integer, which can have value between 0 and 4095.

VIII. Take s string of samples on two analog voltages

This function is invoked by command “**\$B256,8400(CR)**”, where the first argument 256 stands for the number of samples to take in a string, and the second argument 8400 stands for the time interval between samples taken at pins PA2 and PA3. There may be up-to 16000 samples in a string, and the interval is given in clock ticks (8400 == 100µs). The sampling starts immediately, but can take long time depending on the time interval and the number of samples. Once all samples are available the Discovery board will send them to PC. Here first four bytes sent contain high (0) and low (1) bytes of result for pin PA2, sample 0, and high (2) and low (3) bytes of result for pin PA3, sample 0. Next four bytes contain four bytes for sample 1, and so on. The software at the PC should combine these bytes into meaningful results.

IX. Continuously sample two analog input voltages

This function is invoked by command “**\$J8400(CR)**”, where the argument 8400 (8400 == 100 $\mu$ s) stands for the number of clock ticks between consecutive samples taken at pins PA2 and PA3. The samples are taken in blocks of 256, and sent to PC as a packet of 4 x 256 = 1024 bytes immediately after the current block is finished. Samples within the block are packed as explained above (VIII)), and the PC should continuously convert these packs of bytes into meaningful results. The sampling can be stopped by command “**\$K(CR)**”.

X. Count digital pulses

This function is invoked by command “**\$P(CR)**”, and has no arguments. The call initializes a 16 bit counter inside of the microprocessor; the input to the counter is located at pin PB15. The counter counts continuously; after 65535 it rolls back to 0. The user must read the content of the counter by command “**\$R(CR)**”, which returns two bytes forming a 16 bit result. The first byte sent is the LSB, the last is the MSB.

XI. Generate exponential tail pulses as coming from a nuclear detector

This function is invoked by command “**\$L500,1(CR)**”, where the first argument 500 defines the time interval between pulses in microseconds (roughly, for periodic pulses only), and the second argument requests periodic pulses to be generated. The decay time of the pulse is fixed to about 70 $\mu$ s. If the second argument is changed to 2 then the generated pulses are random in time but fixed in amplitude, and the first argument defines the rate; lower number means less pulses per unit of time. The pulses are available at pin PA4. The generation can be stopped by command “**\$D(CR)**”.

XII. Perform digital pulse processing on nuclear pulses and measure their amplitude

This function is invoked by command “**\$N25,10,58,13,9(CR)**”, where the first three arguments represent parameters K, L and pole/zero compensation from article (Digital techniques for real-time pulse shaping in radiation measurements, T. Jordanov, F. Knoll, ..., Nuclear instruments and methods in physics research A 353 (1994) 261-264). The last two arguments are used for averaging of the flat top of the trapezoidal pulse, the fourth argument being the start of the averaging in microseconds, and the last argument the duration of averaging in microseconds. The results are returned as two consecutive bytes for every pulse measured, it is the task of the PC to convert these two bytes in common result representing the amplitude of the measured nuclear pulse and to combine a series of results into a spectrum. The first byte returned is the low part of the result. Nuclear pulses should be applied to pin PC1, and the filtered version is available at pin PA5. The pulse at pin PE6 signals the time of averaging of the trapezoidal pulse.

The digital pulse processing can be stopped by command “**\$O(CR)**”.

## Communication functions in Labview

In order to use the RS232 communication channel the program must first claim its resources by “configuring” the channel. If the requested channel is free to be used then the program can either “write” to or “read” from the opened channel. The program should release the claimed resources before ending, and can do so by “closing” the channel.

a) Open channel

The Labview function to claim and simultaneously initialize the resources of a serial port is shown in Fig. 4.

The function requires parameters (top, left, and bottom side of the box). The function must know which serial port the user wants to

open, and the correct code must be passed into the function at "VISA resource name". Other inputs have default values; baud rate for instance defaults to 9600 baud, as stated in brackets beside the "baud rate" input. The programmer may wire another value to this input or leave it unconnected to use the default. We will wire 921600 to this input to use the baud rate supported by the Discovery board/software. The functions for serial communication can wait some time to execute in case the resource is busy for some reason, and this time ("timeout") defaults to 10 seconds. This seems too much, and we will change it to 1 second by wiring 1 to this input. The receiving or transmitting of characters can be automated in a way that functions search for termination characters within the received string or append termination characters to transmitted string. This will not be needed, and we will cancel this option by connecting logic false to corresponding input ("Enable Termination Char").

The function returns identification code of the claimed serial port ("VISA resource name out"). This code must be used throughout the program whenever one wants to use this serial port, so it must be wired to all functions in the program that use the opened serial port.

The function also returns an error report ("error out"). If an error occurs within any of the serially connected functions then the functions following the one that reported an error cannot work properly, and should be notified that something is wrong. Therefore, the error report must be propagated between functions that use the same serial port. Additionally, the error reported by a function can be intercepted, and some action can be taken to remedy the situation. In our case programs are small and are not intended to be foolproof, so the handling of errors will not be implemented beyond the message to the user that something went wrong.

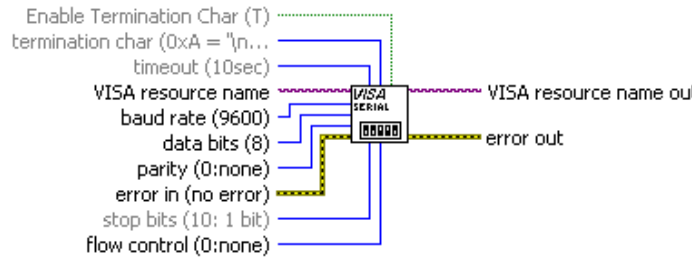


Fig4: A Labview function to »OPEN« the serial port

b) Write

The Labview function to write a string of characters to a serial port is shown in Fig. 5.



Figure 5: A Labview function »WRITE«

The function receives the identification code of the opened serial port (VISA resource name) and an error message (error in) from the preceding function ("OPEN"). The string of characters to be sent over the serial port is wired to an additional input ("write buffer"). Please note that only a string of characters can be wired to the input of this function, if other kinds of data is to be sent, like bytes, integers or floats, these must first be converted to strings. The characters in these strings may include all characters with ASCII codes from 0 to 255, and not only printable characters.

The function passes the identification code of the serial port and a possible error message to the next function. The function also returns the number of characters written to the serial channel.

c) Read

The function to read a string of characters from a serial port is shown in Fig. 6.



Figure 6: A Labview function "READ"

The function receives the identification number of the serial port used and a possible error message from the preceding function ("OPEN").

The function needs additional parameter telling it how many characters to read from the serial port. The function will wait for this number of characters for a predefined time ("timeout", set initially with the function "OPEN"), and return the string of characters at "read buffer". The function will also return the actual number of characters read at "return count". The function will pass the identification number of the serial port used and a possible error message to the next function.

Please note that the string returned consists of individual characters, which might have different meaning than just printable characters. Actually, these characters can have any ASCII value between 0 and 255, and can be interpreted as bytes, integers or some special code. However, the interpretation must be performed using additional conversion functions.

d) Close

Once the writing and reading to the serial port is finished and the program is about to close all captured resources must be released for other programs, therefore the serial channel must be "CLOSED". The function to close the serial channel is shown in Fig. 7.

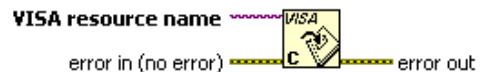


Figure 7: A Labview function "CLOSE"

The function receives the identification code of the serial port used and possible error message. It returns the error code if anything went wrong during the use of the serial port, otherwise it returns an empty message.

A simple program to open a serial port, write a string of characters "ABCD" to it and then closes the port might look like the one at Fig. 8. Similarly, the program to read a string of characters from a port can be something like shown in Fig. 9.

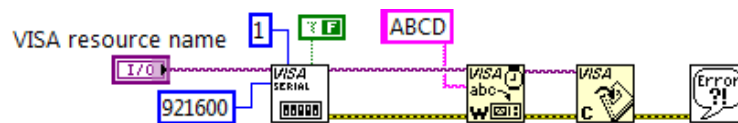


Figure 8: A Labview program to write a string "ABCD" to a serial port using a baud rate of 921600

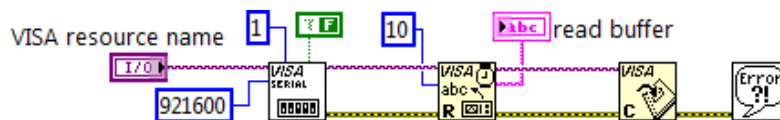


Figure 9: A Labview program to read a string of 10 characters from a serial port and show it at the user interface. Please note: if the string is not available, the program returns an error

There are few additional functions available in Labview to handle the serial communication.

e) Bytes available

We might not know in advance how many characters are available in the serial port buffer but we need to tell the function for reading how many characters to read. Should we wait some more



time before the characters accumulate in the input buffer? How much time? Should we read as many characters as available and return later to check for the rest? Whatever our intentions are we need to know how many characters are presently available, and a Labview has a function for this. It is shown in Fig. 10.

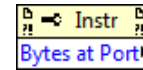


Figure 10: A Labview function »Bytes available«

The function requires two inputs; these are the identification number of the serial port and a possible error message. Both are passed to the output for succeeding functions. Additionally, the function returns the current number of characters waiting in the input buffer.

A sample program to wait until ten characters are available in the input buffer and then reads them is shown in Fig. 11.

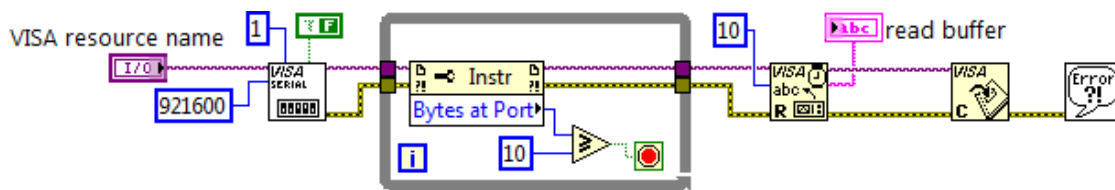


Figure 11: A simple program opens a serial port, waits in the loop until ten characters are available, then reads them, displays and closes the serial port

f) Flush buffer

Characters may accumulate in the input buffer of a serial port if not read on time. If our program needs to start reading characters now, but does not know about leftovers in the input buffer, it will read the leftovers first and this is not what we need. It would be wise to get rid of whatever is in the buffer first and only then proceed with the read. The Labview offers a function called "VISA Flush I/O buffer", which does exactly this: it clears buffers of any leftovers and allows fresh accumulation of characters. The function is shown in Fig. 12.



Figure 12: A Labview function »VISA Flush I/O buffer«

## Ways to communicate with a peripheral using a serial port

A typical communication between a PC and a peripheral is similar to a human dialogue where the first person asks a question, and the second person answers. Here the PC sends a command to the peripheral, and the peripheral executes the command and/or returns an answer. The commands our test peripheral understands were listed and detailed above.

a) A simple send example

In order to change the state of digital outputs at the Discovery board the command III) should be issued ("H##(CR)"); the '##' stands for a decimal number between 0 and 15. The program to change the state of the digital outputs might look like the one in Fig. 13.

The program starts by opening the serial port, and proceeds to writing the desired string to the serial port. The string is constructed by concatenating three strings: "H", the formatted user input "Dout", and the (CR) character. Finally the serial port is closed and eventual error message presented to the user.

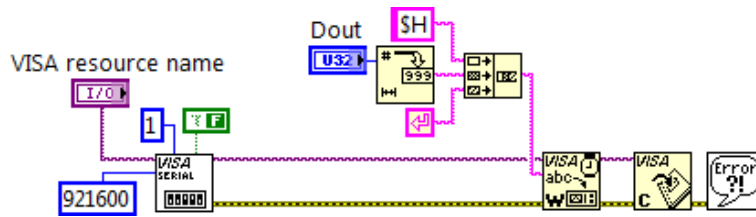


Figure 13: A simple program to send string »\$H##(CR)« to the Discovery board.

b) A simple send/wait/receive example

In order to receive an introductory string of characters from a Discovery board the PC must send character '\$' to the board (command I), then wait some time and read the string from the serial port. The Labview program might look like the one in Fig. 14.

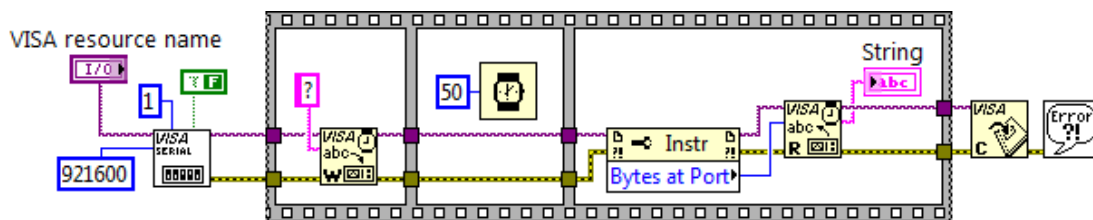


Figure 14: A simple program to send/wait/receive a string of characters

The program starts by opening the serial port with the specified parameters for port number and baud rate, and proceeds into a “film strip” part. The important property of this part is that the windows in the strip execute sequentially, that is whatever is put into the leftmost window of the “film strip” gets executed first, and so on until the rightmost window in the strip.

Within the “film strip” first a character '?' is sent through the serial port to the Discovery board using the “WRITE” function in the leftmost window. Following this the execution proceeds to the middle window where it waits for 50ms. After the delay the execution continues to the rightmost window of the strip, where the number of bytes waiting in the input buffer is retrieved. The returned number of bytes is used as an input to the function “READ” to define the number of characters to read from the serial port. The characters read from the port are then presented to the user using a simple string indicator, and the serial port is closed.

c) More complex send/wait/receive example with decoding

The analog to digital converter on a Discovery board is programmed to return four bytes in a form of a string of characters on command VII) (“\$A(CR)”). These characters must be converted to bytes and bytes combined to integers to get usable results. A Labview program to accomplish this might be the same as shown in Fig. 15.

The program starts with opening the serial port and continues with sending a character string “\$A(CR)” to the Discovery board. The program must then wait some time for results from the Discovery board, and the middle window within the film strip causes the waiting. Within the leftmost window of the film strip a string of four characters is read from the serial port (the four characters better be there or-else an error will be issued). The string is then converted to an array of bytes, which is essentially the same as taking an ASCII code of each character individually and combining these characters into an array in other programming environments. The individual bytes are then extracted from the array, combined by pairs into two integer numbers, and presented to the user at indicators “PA2” and “PA3”.

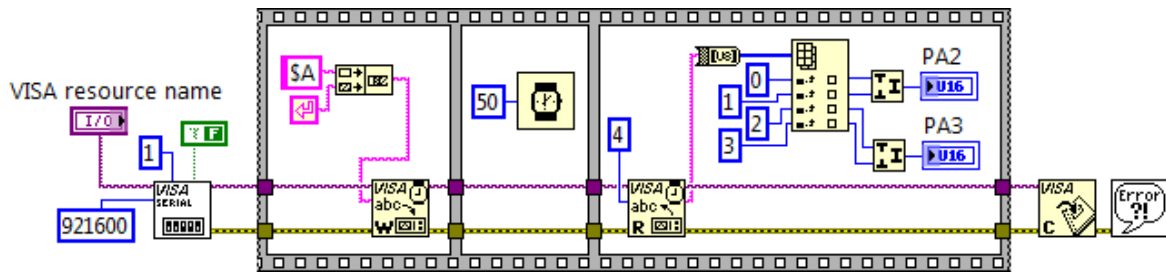


Figure 15: A Labview program to read and interpret bytes

d) Avoiding the fixed waiting loop

When we expect a longer string of characters, like when we request the string of measurements (VIII)), and we do not know exactly how long it will take for the characters to appear, we cannot simply use the fixed delay in the middle window of the film strip like in previous example. A variable delay is preferred. Better still: since we know how many characters to expect, we can use the function “bytes at port”, and simply poll this function until it reports the required number of characters in the input buffer before we proceed to the rightmost window of the film strip. The program might look like the one in Fig. 16.

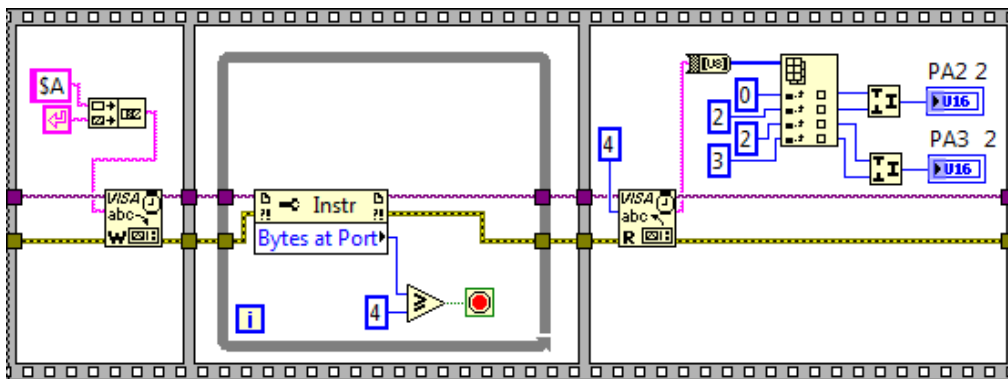


Figure 16: A program to read and interpret bytes without the fixed waiting;  
note that the opening and closing of the serial port is not shown

The program will now loop in the middle window of the film strip until there are four characters available in the input buffer. This may seem too long in case something goes wrong and the Discovery board does not send any characters. It would be wise to limit the number of loop iterations, and report an error if characters are not received within the time expected.

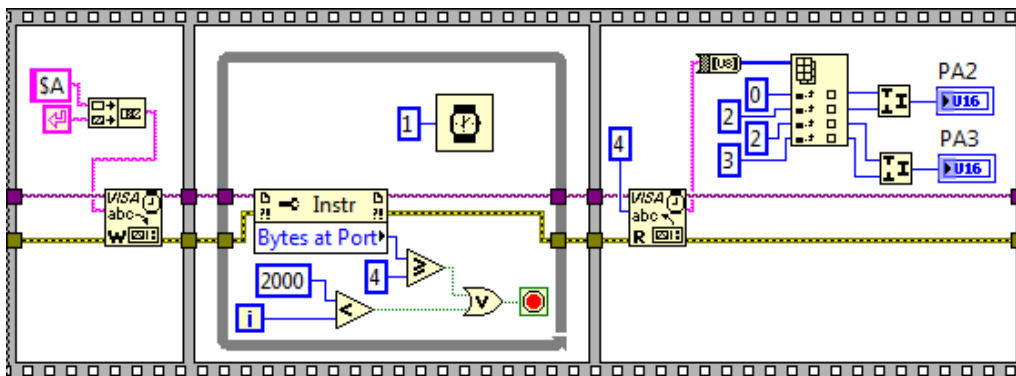


Figure 17: We can wait for the string of four characters until they arrive, but not more than 2000 x 1ms

Additionally, such a loop repeats very fast inside of the PC computer, and eats-up all resources slowing down the execution of concurrently running programs. It is wise not to poll the function “bytes at port” so very often, once per millisecond should suffice. The best is to insert a “wait ms” function into a while loop and slow down its repeating. In this case it is also simpler to limit the number of iterations since we now know that each iteration of the while loop takes one millisecond. The improved version is shown in Fig. 17.

e) Wrap the above programs together - polling

We have an interface (the Discovery board) which understands different commands to accomplish different tasks. We need a program where a user will select what the interface is supposed to do. We shall prepare a user interface containing several buttons, and each of the buttons will start one of the subprograms from above when clicked. The front panel of such a program is shown in Fig. 18. Since we want a subprogram to be started only once when a user clicks a button, the “Mechanical action” for these buttons is selected as “Latched when pressed”. The background of these buttons can be colored differently than the front to confirm the click.

A while loop can be used to continuously poll these buttons. Once a certain button gets clicked, a corresponding subprogram is run. Therefore the while loop should enclose the three subprograms and buttons. The Labview program is given in Fig. 19.

The program starts with the opening of the serial port (not shown in Fig. 19), and then proceeds into the while loop. The identification number and error message obtained during the opening here branches to three case structures. The upper case structure contains the first subprogram to send digital data to the Discovery board, the middle contains program to inquire about the introductory string, and the lower contains program to perform a measurement on the ADC and show results on the PC screen. Each of these case statements is invoked when a corresponding button at the user interface gets clicked. Note that the button once clicked remains in active state until the program reads it, then it returns to the initial state due to the mechanical action selected.

The while loop is stopped when user clicks the “Stop” button, and then the closing of the serial port and possibly the displaying of the error message takes place (not shown in Fig. 19). The “wait ms” function is inserted into the while loop to slow-down its execution and save PC resources.

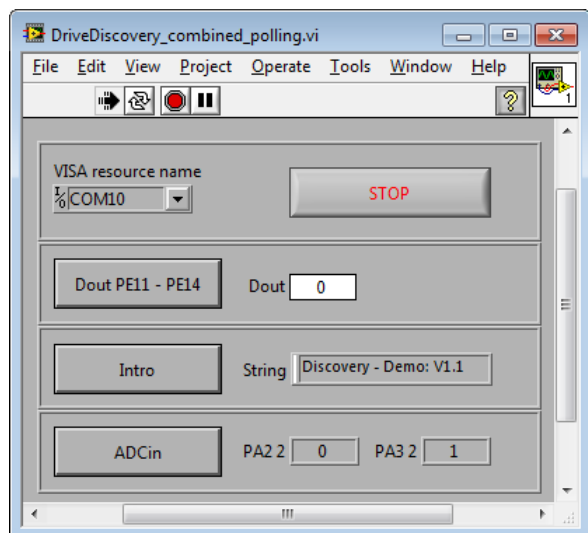


Figure 18: The front panel of the combined program

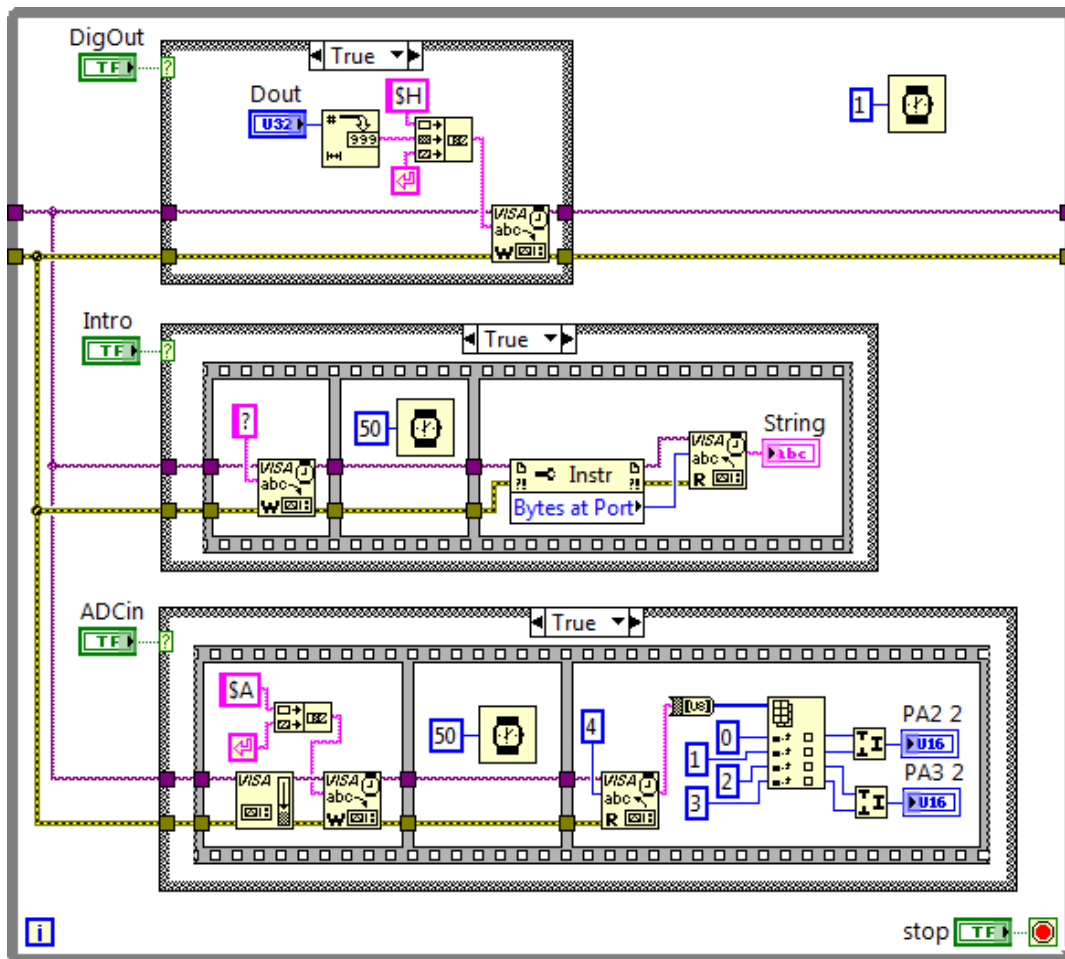


Figure 19: Polling is used to wrap the three subprograms together; the opening and closing of the serial port is not shown, but should be executed before the entrance to the while loop and after the exit of the loop, connectors are shown on the figure

f) Wrap the above programs together – events

Continuous polling of buttons is interrupted during the time any of the subprograms is executed. Consider what happens if we need to wait for the string from the Discovery board for five seconds. You can test this by changing the input to the “wait ms” function in the middle case statement to 5000. The loop does not respond to any clicks when it is busy executing one of the subprograms! It does, however, memorize any clicks and executes them after the execution of the lengthy case structure is finished. We shall use different technique to avoid such problems.

The Labview implements event handling. An event is, for instance, a click on a button. The program can be written to simply wait, and only when the user clicks a button a certain subprogram gets executed.

The event structure should be used inside a while loop, one event structure per loop. Events must be first registered with the structure, and one window inside the structure is reserved for each registered event. Whatever we put inside the window will be executed on this event.

The above program rewritten with the event structure is shown in figures 20 to 23. There are four figures to show programs for four different events, since there are four buttons on the user interface panel. The opening and closing of the serial port is not shown.

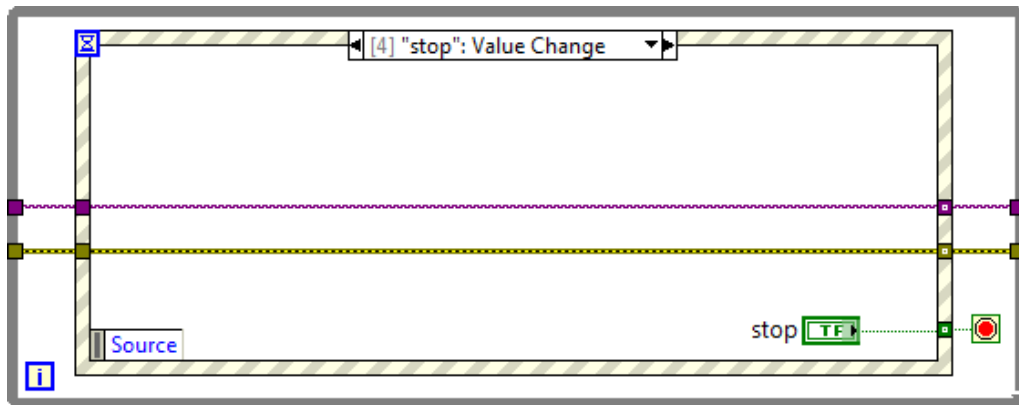


Figure 20: The event structure with a window to handle the click on a button »stop«

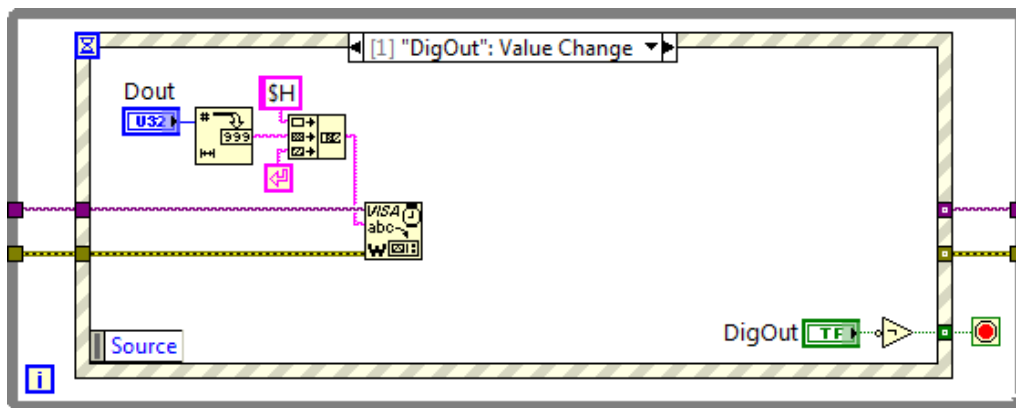


Figure 21: The event structure with the window to handle the click on the button »DigOut«

The program enters the while loop after the opening of the serial port. The while loop does not contain any executable functions, so it simply waits to be stopped. It is empty and does not load the processor; you can confirm this by opening the Windows Task Manager/Performance. The Fig. 20 shows what is to be executed when the button “stop” is clicked. The value of the button should be passed out of the event structure to stop the execution of the while loop and stop the program. Additionally, the identification number and error message of the serial port are to be passed to function for closing of the port.

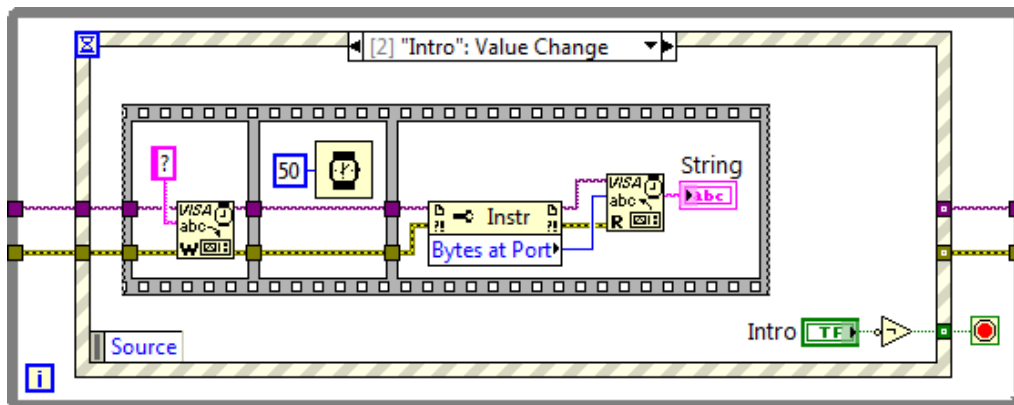


Figure 22: The event structure with the window to handle the click on the button »Intro«

Similarly, when the button "DigOut" gets clicked, the event structure shown in Fig. 21 gets executed. Here the "\$H##(CR)" gets sent to the Discovery board. The button that caused this event must be read within the window handling this event, and the value of button "DigOut" is read (as "true" since the button was clicked), inverted, and passed to the while loop preventing the end of its execution.

Figures 22 and 23 respectively show what is to be executed when buttons "Intro" and "ADCin" get clicked. Note that the value of the button must be read within the corresponding event window; its value is first inverted and then passed to the while to prevent stopping its execution.

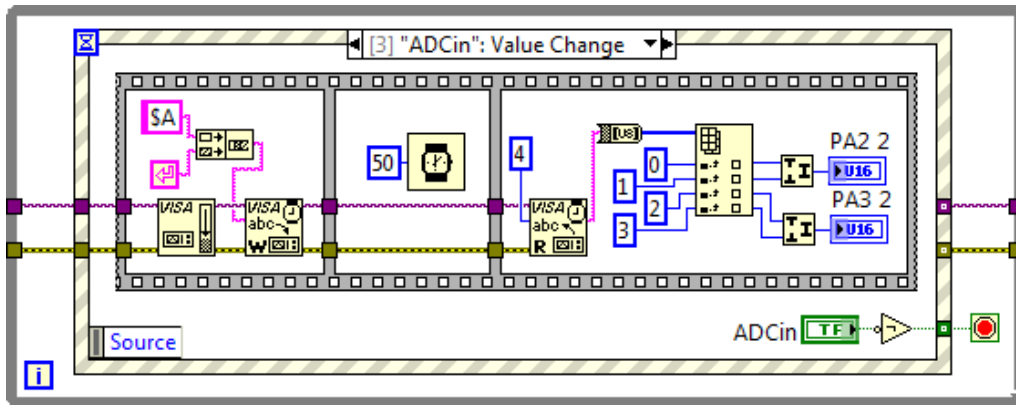


Figure 23: The event structure with the window to handle the click on the button »ADCin«

The use of event structure as shown above simplifies the programming, but does not solve the problem of waiting inside of one of the windows in the event structure. You can confirm this by changing the delay in "Intro" window; the next event is accepted, but causes the execution only

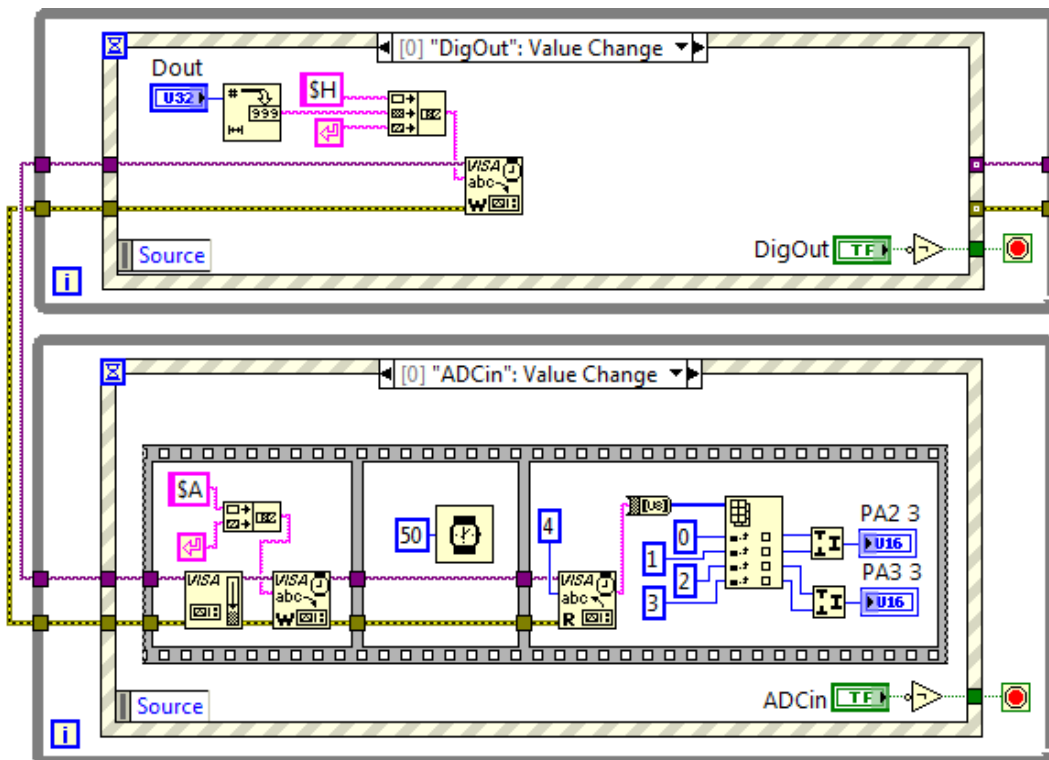


Figure 24: Two processes are created by drawing two while loops; each loop contains an event function. Upper loop handles clicking on buttons »DigOut« (shown here) and »stop«, lower loop handles events caused by clicking on buttons »ADCin« (shown here), »Intro« and »stop«.

after the currently running event is finished. The point here is that one while loop creates one process within the PC, and one process can handle one event at a time, so next event must wait.

To improve the situation more processes (more while loops with event processes) can be created. Consider the program in Fig. 24. It is customary to distribute events among the processes in such a way that they do not interfere with each other. In our case buttons "Intro" and "ADCin" are not expected to be clicked simultaneously, since the Discovery board cannot respond to both commands simultaneously anyway. The handling of these two events can be joined into one process. It is however possible to request the ADC sampling and simultaneously ask for the change at the digital outputs, the Discovery board can handle both; therefore clicks on these two buttons should be handled by two separate processes. The click on the button "stop" should be handled immediately, since it does not interfere with the Discovery board in any way; this should then be a separate process. However, clicking of the button "stop" does not interfere with changing of the digital outputs of the Discovery board as well, so it can be handled by the same process. There should be therefore two processes, one for buttons "stop" and "Dout", and one for buttons "Intro" and "ADCin". The usual practice is to limit the number of processes as much as possible to keep the program simple. The program is shown in figures 24 to 26.

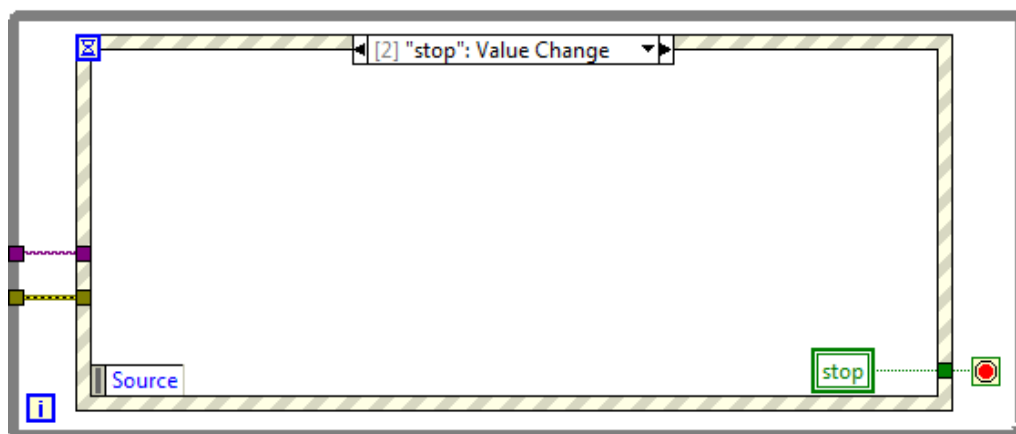


Fig 25: Lower loop with event handling for button »stop«

Note that local variables representing the button "stop" are used in the lower loop; these local variables are needed to notify the lower loop when to stop the execution. Note that the delay need to wait for the string of characters was broken into several shorter delays, and these were used in a loop (Fig. 26, lower loop, middle window in the film strip). The looping is broken when the user clicks the button "stop", so there is no noticeable delay in ending the program.



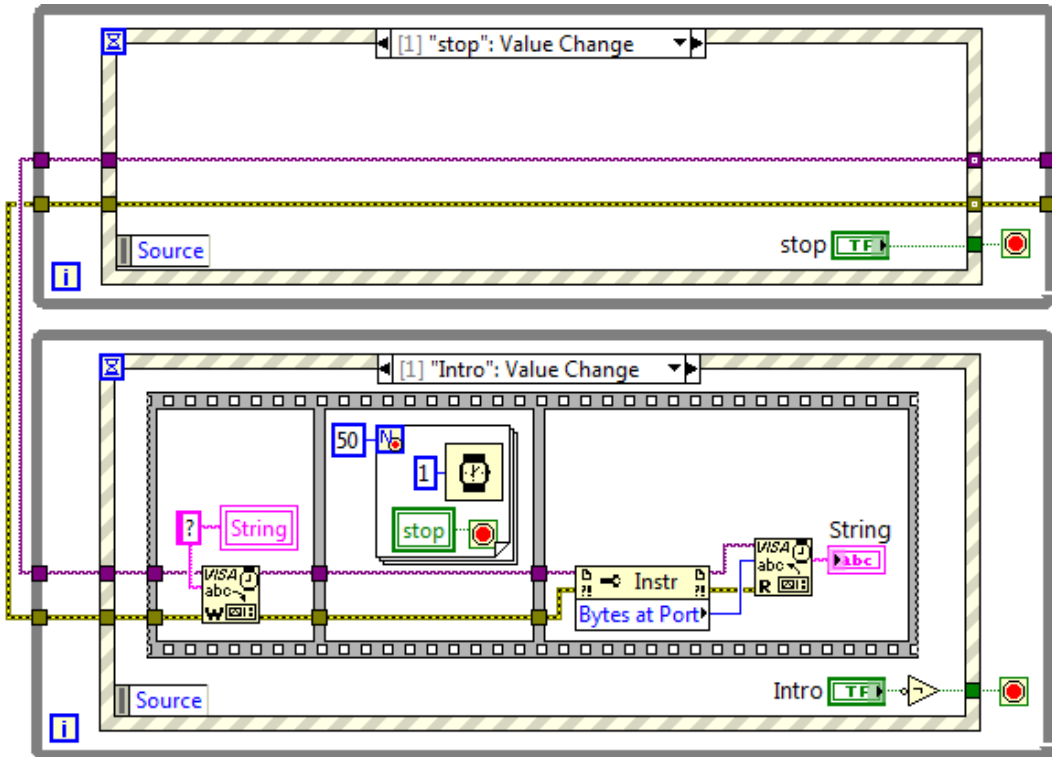


Figure 26: Upper loop with event for clicking »stop« button, lower loop with event for clicking »Intro« button