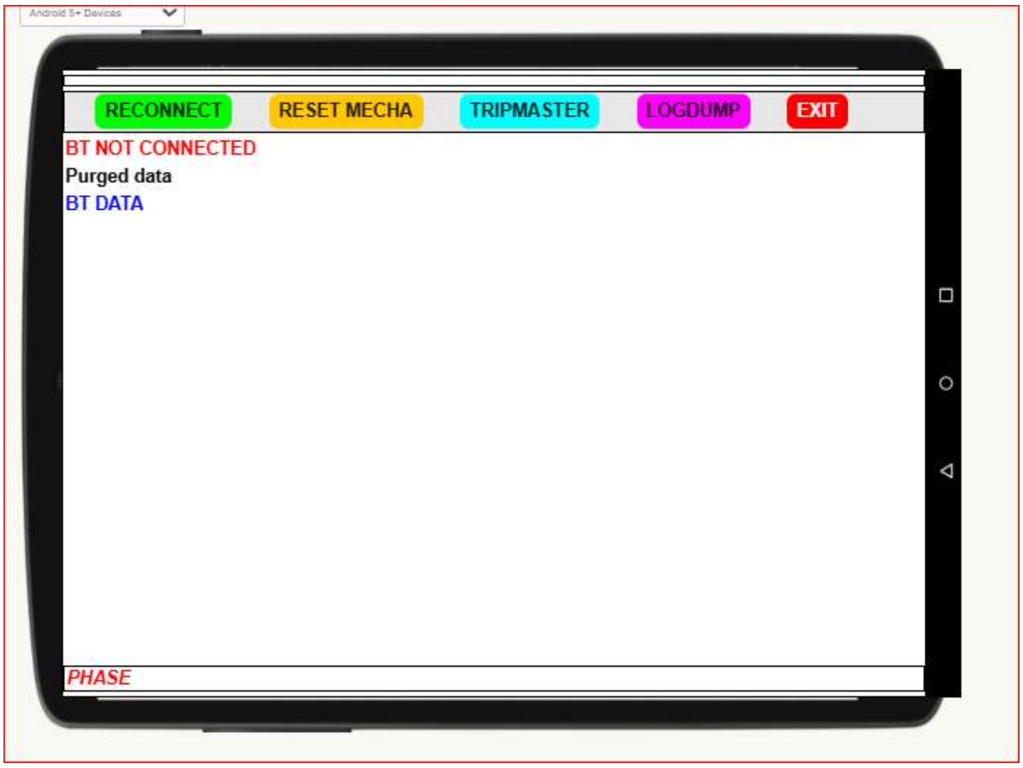
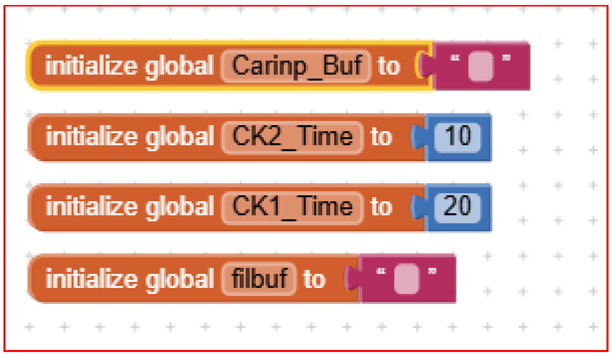
In the following pages it is explained how a classic BT client (Tx and Rx) can be managed by Itoo in an AI2 app. The purpose is to handle a full-duplex communication between the app and an ESP32 device, capable of receiving fast frames from the ESP32 and, contemporary, to send commands.
The communication is safeguarded by a watchdog, that is capable to restore the communication when, for any reason, the communication fails or ceases.
One of the advantages of this approach is to leave free the "foreground" activities, i.e. those directly managed by AI2, from those inherent only to the BT, with the possibility to handle all the buttons, the labels, in one word: the UI.
The steps that describe the flow are:

1. Initialize the Screen1 with some stuff
2. Initialize the Itoo background process in which the BT client is invoked and a clock timer event to send/receive data on the BT radio is linked to Itoo as well.
3. After a while (i.e. 2 seconds) verify whether the BT client has been successfully started.
4. From the point onward the BT communication is managed in background and the foreground can do its job parsing the data received on the BT, if any, and sending data on the BT, just by loading a buffer that will be sent out by the clock in background.

The following picture depicts how the main screen looks like.



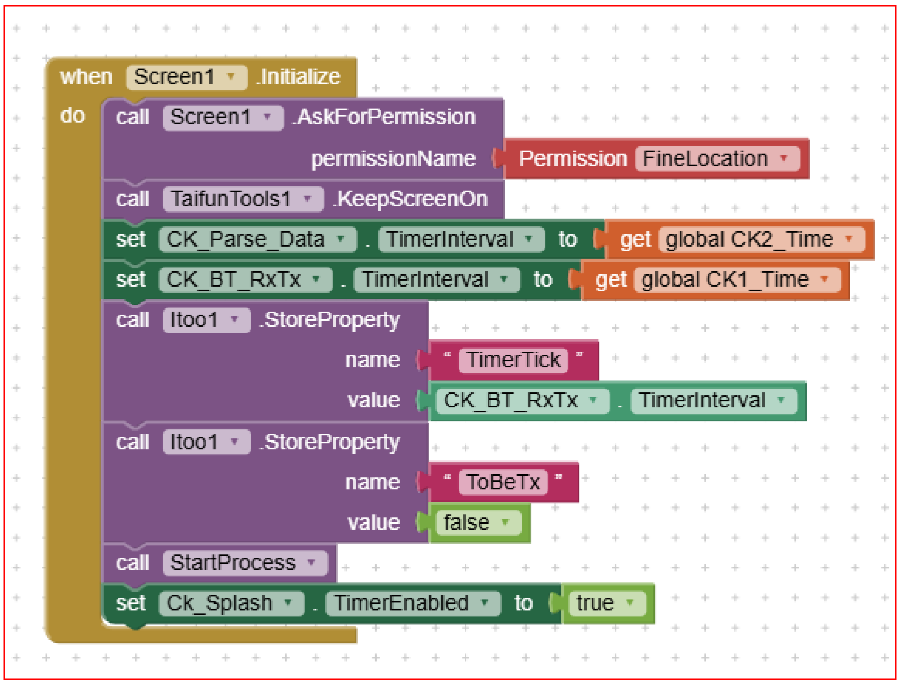
The main variables:


```
initialize global Carinp_Buf to " "
initialize global CK2_Time to 10
initialize global CK1_Time to 20
initialize global filbuf to " "
```

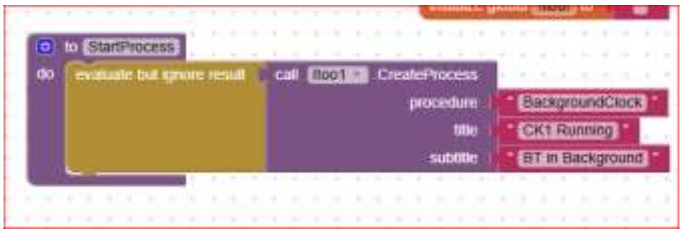
Carinp_Buf: buffer to contain the data received (text format)
Filbuf: buffer to scontain data to be stored into a file (text format)
CK1_Time: time constant of the receiving service (20 milliseconds)
CK2_Time: time constant of the parsing function. **NOTE** it must be faster of CK1 (to have a double speed it must be the half of CK1) in order to avoid loosing data.

In the Screen1 initialization :

- the FineLocation permission is asked to allow the BT to work
- the Taifun's Tool (credits !!!) extension is used to maintain the screen always alive
- The two clocks time constant properties are loaded into their respective clocks
- The Ck1 (= CK_BT_RxTx) TimeInterval is passed to Itoo using the StoreProperty method because it is a background clock
- The CK2 one (= CK_Parse_Data) is loaded directly because it is a foreground clock
- The flag instructing Itoo that no characters have to be sent, for the time being, is set (false)
- Itoo is started
- To allow time for Itoo to start, a splash screen clock is started (2 seconds, for example)
  this means that for two seconds no other operations are executed waiting for Itoo to become operative

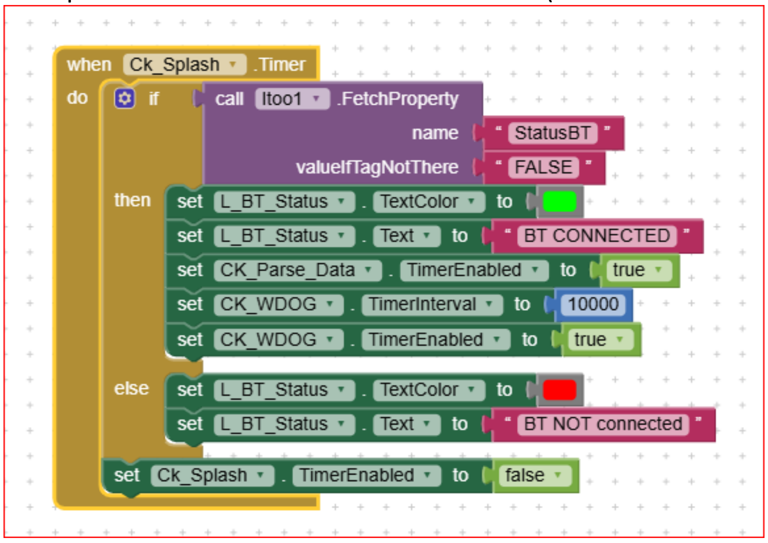

```
when Screen1 .Initialize
do  call Screen1 .AskForPermission
                     permissionName  Permission FineLocation
    call TaifunTools1 .KeepScreenOn
    set CK_Parse_Data . TimerInterval to  get global CK2_Time
    set CK_BT_RxTx . TimerInterval to  get global CK1_Time
    call Itoo1 .StoreProperty
                  name  " TimerTick "
                  value  CK_BT_RxTx . TimerInterval
    call Itoo1 .StoreProperty
                  name  " ToBeTx "
                  value  false
    call StartProcess
    set Ck_Splash . TimerEnabled to  true
```

Itoo background process is started by loading the name of the procedure that will run in background.
Title and subtitle are shown for a while in a popup window so to advise that Itoo is started. The latest version of the extension does not show any longer the popup, this is useful if you need to start Itoo silently.



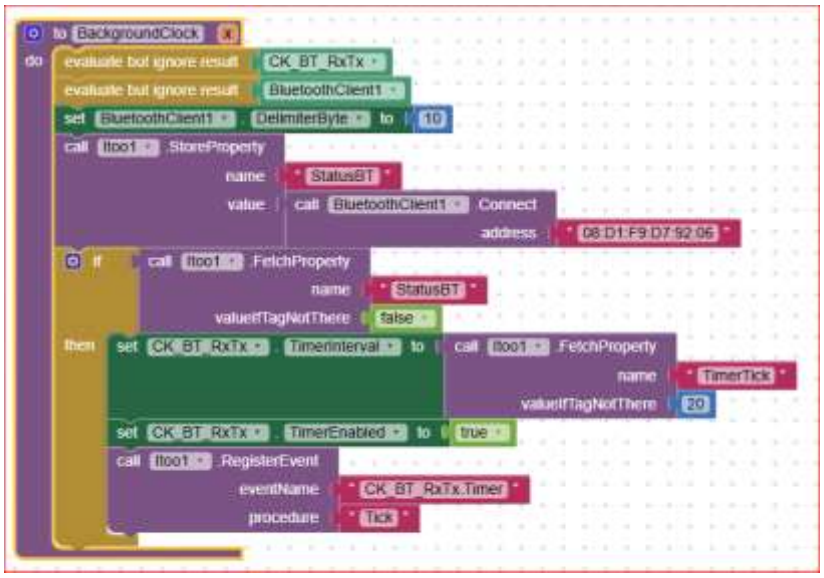
```
to StartProcess
do  evaluate but ignore result   call Itoo1 .CreateProcess
                                          procedure  "BackgroundClock"
                                              title  "CK1 Running"
                                           subtitle  "BT in Background"
```

As said before, the splash screen is a procedure, triggered by a clock, which will run some time after the CreateProcess is invoked, so to allow the BT communication to be activated (otherwise the property BT_Status could be not set, yet).
If the BT status is ok (true) this means that the BT has been successfully connected and is ready to work. In this case the watchdog timer is firstly set to 10 seconds (later on it will be set to a faster intervention time).
The splash screen clock is then auto-disabled (i.e. it shall run only once).



```
when Ck_Splash .Timer
do  if   call Itoo1 .FetchProperty
                             name  "StatusBT"
                valueIfTagNotThere  "FALSE"
    then  set L_BT_Status . TextColor to 🟩
          set L_BT_Status . Text to "BT CONNECTED"
          set CK_Parse_Data . TimerEnabled to true
          set CK_WDOG . TimerInterval to 10000
          set CK_WDOG . TimerEnabled to true
    else  set L_BT_Status . TextColor to 🟥
          set L_BT_Status . Text to "BT NOT connected"
    set Ck_Splash . TimerEnabled to false
```

The BackgroundClock procedure is the Itoo core (please note the “x” parameter that has to be set mandatorily, though it is not used).
Two components are made available to Itoo: the BT_Rx_Tx clock and the Bluetooth Client.
Both are evaluated carelessly of their results.
The Linefeed Character (10 or 0x0A) is set as string terminator for incoming BT data frames. **Note** that it shall be done here in background.
The BT client is tried to be connected (the shown address is that of my ESP32, but in your application it shall be the one of YOUR system) and, if successfully connected, the StatusBT property will be set accordingly: True = ok; False = not connected.
If the BT connection is OK, then the time constant for the CK_BT_RxTx is fetched from the foreground (as per Itoo foreground⇔ background data passing method) and stored into the Clock Time Interval property. If the value is not available, a default 20 millisecond value is passed, anyway.
Finally, the CK_BT_RxTx timer is enabled and its event is registered by Itoo, so when it fires (i.e. the Clock.Timer Event is triggered) its service function, named Tick, is handled in the background.

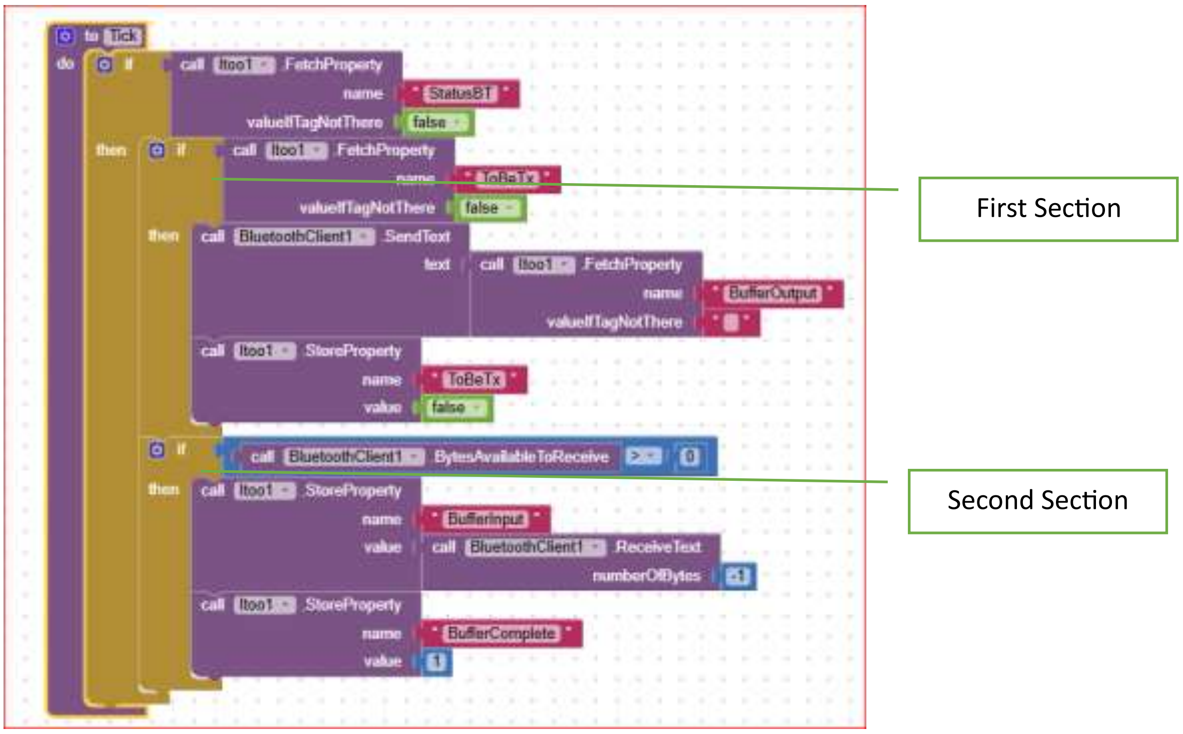


```
to BackgroundClock x
do  evaluate but ignore result  CK_BT_RxTx
    evaluate but ignore result  BluetoothClient1
    set BluetoothClient1 . DelimiterByte to 10
    call Itoo1 .StoreProperty
                   name  "StatusBT"
                   value  call BluetoothClient1 .Connect
                                         address  "08:D1:F9:D7:92:06"
    if    call Itoo1 .FetchProperty
                      name  "StatusBT"
          valueIfTagNotThere  false
    then  set CK_BT_RxTx . TimerInterval to  call Itoo1 .FetchProperty
                                                          name  "TimerTick"
                                                valueIfTagNotThere  20
          set CK_BT_RxTx . TimerEnabled to true
          call Itoo1 .RegisterEvent
                   eventName  "CK_BT_RxTx.Timer"
                   procedure  "Tick"
```

The background timer event is composed by two sections: the first is devoted to send and the second to receive data on the BT radio. Both sections are subjected to a good functioning of the BT client that is retrieved by the property Status_BT.

If a character, or a string, has to be sent, the flag “ToBeTx” is set be the calling procedure and fetched here. If it is True, this means that a string, or a character, shall be sent, and in this case it is fetched from the BufferOutput property (as per Itoo foreground⇔background data exchange method). Once the data has been transmitted, the flag is reset so to avoid further transmissions of the same data.

NOTE please take care that the sending procedure (the first section of Tick) is activated every 20 milliseconds, therefore if the foreground asks faster transmissions, that is: the gap between two consecutive calls of the BT_Send_Text foreground procedure is less than 20 milliseconds, the latter overwrites the first, which is then lost. So if you want to send multiple messages, you can “join” together, as far as possible, the messages, or you can delay the latter messages each of a time about 20 milliseconds, by means of a dedicated clock.



```
to Tick
do  if   call Itoo1 .FetchProperty
              name "StatusBT"
              valueIfTagNotThere false
    then if   call Itoo1 .FetchProperty
                   name "ToBeTx"
                   valueIfTagNotThere false
         then call BluetoothClient1 .SendText
                   text call Itoo1 .FetchProperty
                             name "BufferOutput"
                             valueIfTagNotThere " "
              call Itoo1 .StoreProperty
                   name "ToBeTx"
                   value false
         if   call BluetoothClient1 .BytesAvailableToReceive > 0
         then call Itoo1 .StoreProperty
                   name "BufferInput"
                   value call BluetoothClient1 .ReceiveText
                              numberOfBytes -1
              call Itoo1 .StoreProperty
                   name "BufferComplete"
                   value 1
```

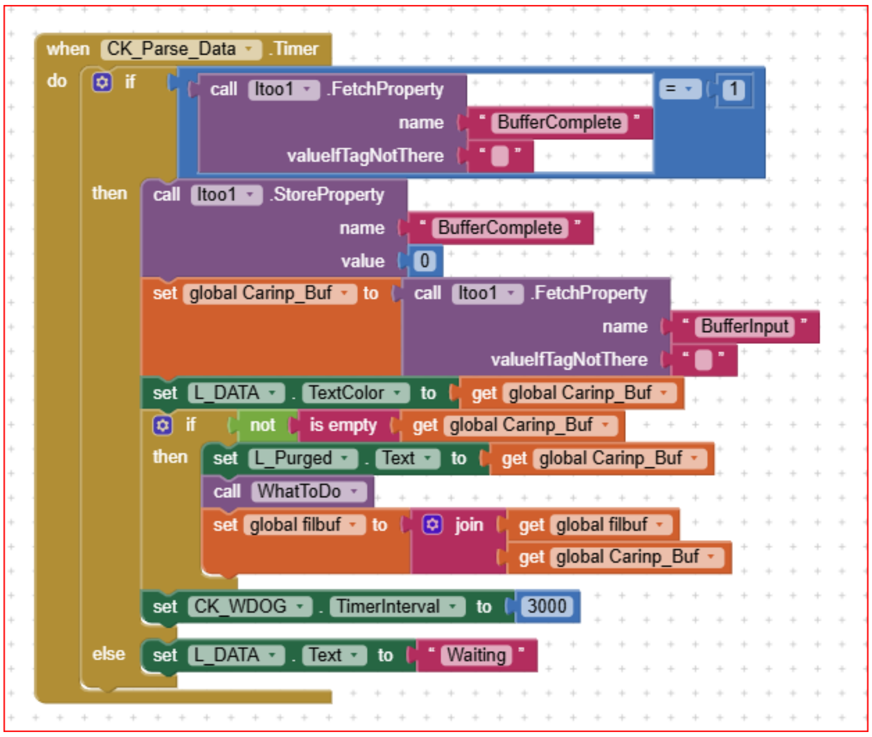
The second section is dedicated to the receiving of data (text data type). The method is the BT client standard one: if characters are available, then the client fetches the incoming data until it encounters a Linefeed character (the one set as terminator). Once the incoming data have been all collected (i.e. until the Linefeed), the BufferComplete property is set to True, to flag to the parsing procedure that data are available.

The Parsing procedure is activated every half time of the receiving in order to be sure to not miss any data frame. If a complete frame has been received by Tick, the flag BufferComplete is set to 1, therefore the frame is parsed to detect what are the data just received.

Otherwise, if a complete frame (i.e. terminated by a Linefeed) is not received, the procedure shows "Waiting" and exits immediately.

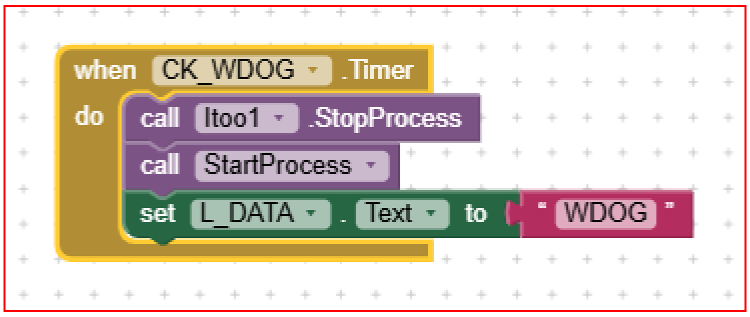
The input buffer is then moved from the Itoo property Buffer_Input to the global variable Carinp_Buf. The raw data is shown in a dedicated label (L_DATA), while the good frame (terminated by a Linefeed and not empty) is shown in another label (L_Purged): in this way in case of a corrupted frame received, it is anyway shown in the L_DATA label, to allow helping in the corruption reason detection.

A the end, the WhatToDo procedure is basically a "switch...case" structure that allows to execute different tasks in relation to the received frame. Whenever a good frame has been received the WDOG clock is retriggered with a time constant of 3 seconds



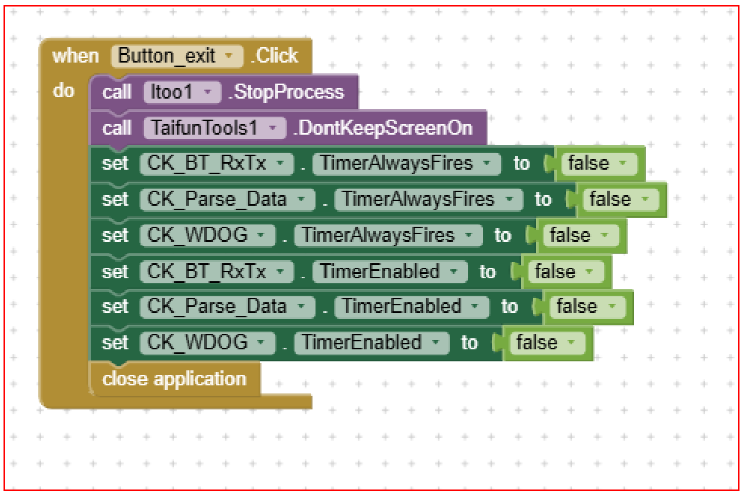
```
when CK_Parse_Data .Timer
do  if    call Itoo1 .FetchProperty
                name "BufferComplete"
                valueIfTagNotThere " "
          = 1
    then  call Itoo1 .StoreProperty
                name "BufferComplete"
                value 0
          set global Carinp_Buf to  call Itoo1 .FetchProperty
                                          name "BufferInput"
                                          valueIfTagNotThere " "
          set L_DATA . TextColor to get global Carinp_Buf
          if   not is empty get global Carinp_Buf
          then set L_Purged . Text to get global Carinp_Buf
               call WhatToDo
               set global filbuf to join get global filbuf
                                         get global Carinp_Buf
          set CK_WDOG . TimerInterval to 3000
    else  set L_DATA . Text to "Waiting"
```

The Watchdog procedure is retriggered every time a good frame is received or every time a character, or a frame, is transmitted. If none of these two conditions happen (good Rx or a Tx) the watchdog is not retriggered, therefore its .Timer event is raised, which produces a stop process, and retarts Itoo again. In this way the BT client is reset also, and a clean, new communication restarts.



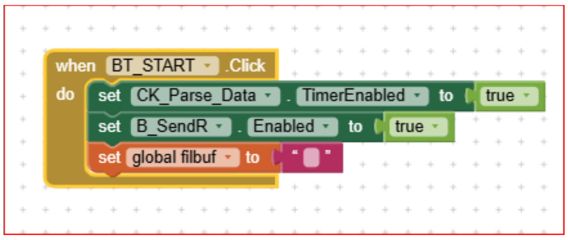
```
when CK_WDOG .Timer
do  call Itoo1 .StopProcess
    call StartProcess
    set L_DATA . Text to "WDOG"
```

To be sure that, when exiting the app, the Itoo background process is terminated correctly, the process is stopped, and all clocks are deactivated as well.



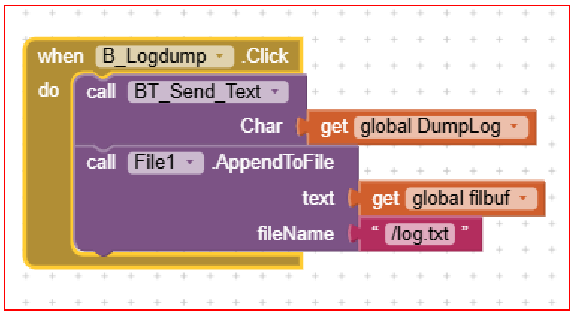
```
when Button_exit .Click
do  call Itoo1 .StopProcess
    call TaifunTools1 .DontKeepScreenOn
    set CK_BT_RxTx . TimerAlwaysFires to false
    set CK_Parse_Data . TimerAlwaysFires to false
    set CK_WDOG . TimerAlwaysFires to false
    set CK_BT_RxTx . TimerEnabled to false
    set CK_Parse_Data . TimerEnabled to false
    set CK_WDOG . TimerEnabled to false
    close application
```

The Button BT_START (re)enables the Parse_Data clock, enables the SendR button (without particular use) and clears the data buffer for the file storage. The Filbuf buffer contains all the messages received by the app, until the button B_Logdump is hit.



```
when BT_START .Click
do  set CK_Parse_Data . TimerEnabled to true
    set B_SendR . Enabled to true
    set global filbuf to " "
```

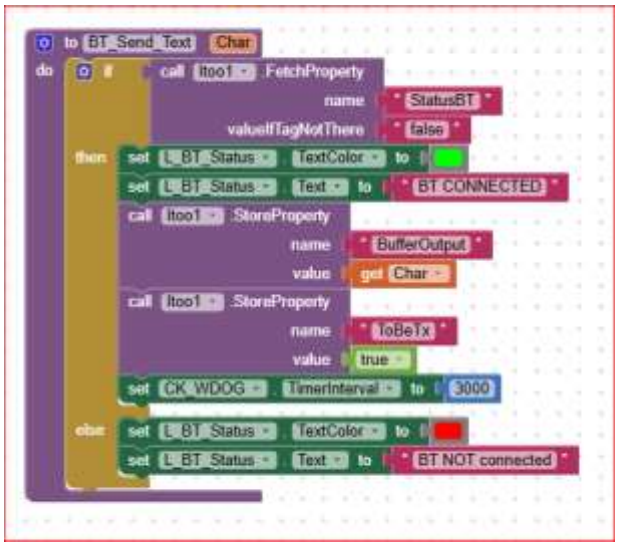
When the B_Logdump button is hit, a Character is sent to the ESP32, and the entire filbuf buffer is stored into the file named log.txt into the ASD. This is only for debugging purpose: by storing everything has been received, one can detect any “weird” nessage could have been received.



```
when B_Logdump .Click
do  call BT_Send_Text
              Char  get global DumpLog
    call File1 .AppendToFile
              text  get global filbuf
          fileName  " /log.txt "
```

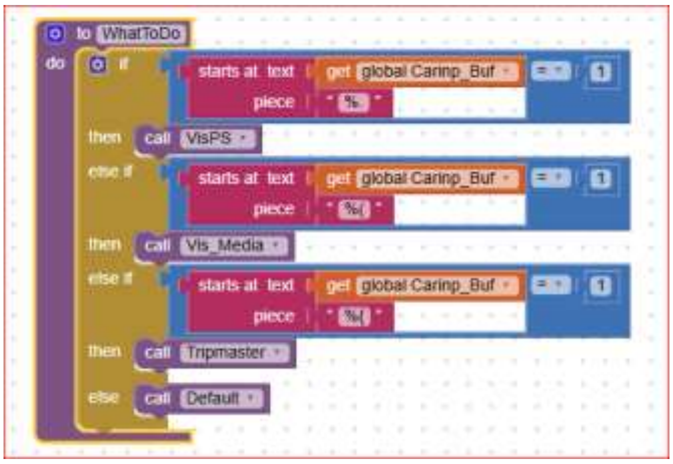
The Send_Text procedure is a foreground one, and is called whenever the foreground wants to send a text (one char or string). It operates only if the BT_Status is True. The data to be transmitted is received through the Char parameter, then is stored into the Buffer_Output property, from which the Tick procedure will fetch the contents. To allow the transmission the flag ToBeTx is set to True and it is reset to False, after the transmission, by the Tick background procedure.

After all this has been done, to maintain silent the watchdog, it is retriggered by loading the time constant to its default (3 seconds) value. This is necessary because if the transmission of one, or more texts last more than the standard 3 seconds, the watchdog can erroneously fire even though the transmission is in progress.



```
to BT_Send_Text Char
do  if    call ltoo1 .FetchProperty
                     name "StatusBT"
          valueIfTagNotThere "false"
    then  set L_BT_Status . TextColor to 🟩
          set L_BT_Status . Text to "BT CONNECTED"
          call ltoo1 .StoreProperty
                     name "BufferOutput"
                     value get Char
          call ltoo1 .StoreProperty
                     name "ToBeTx"
                     value true
          set CK_WDOG . TimerInterval to 3000
    else  set L_BT_Status . TextColor to 🟥
          set L_BT_Status . Text to "BT NOT connected"
```

The WhatToDo procedure is like a switch…case structure intended to start specific tasks depending on the message being received. The Heading characters of the message are used to discriminate the task. In this example the tasks are just for information only (i.e. they are empty). The sample ESP32 code supplied with this demo, sends to the app three types of strings. This is intended to show the capability to discriminate the headers and therefore the "Phase" label is set accordingly while the app is running.



```
to WhatToDo
do  if       starts at  text get global Carinp_Buf   = 1
                       piece "%"
    then     call VisPS
    else if  starts at  text get global Carinp_Buf   = 1
                       piece "%("
    then     call Vis_Media
    else if  starts at  text get global Carinp_Buf   = 1
                       piece "%{"
    then     call Tripmaster
    else     call Default
```