

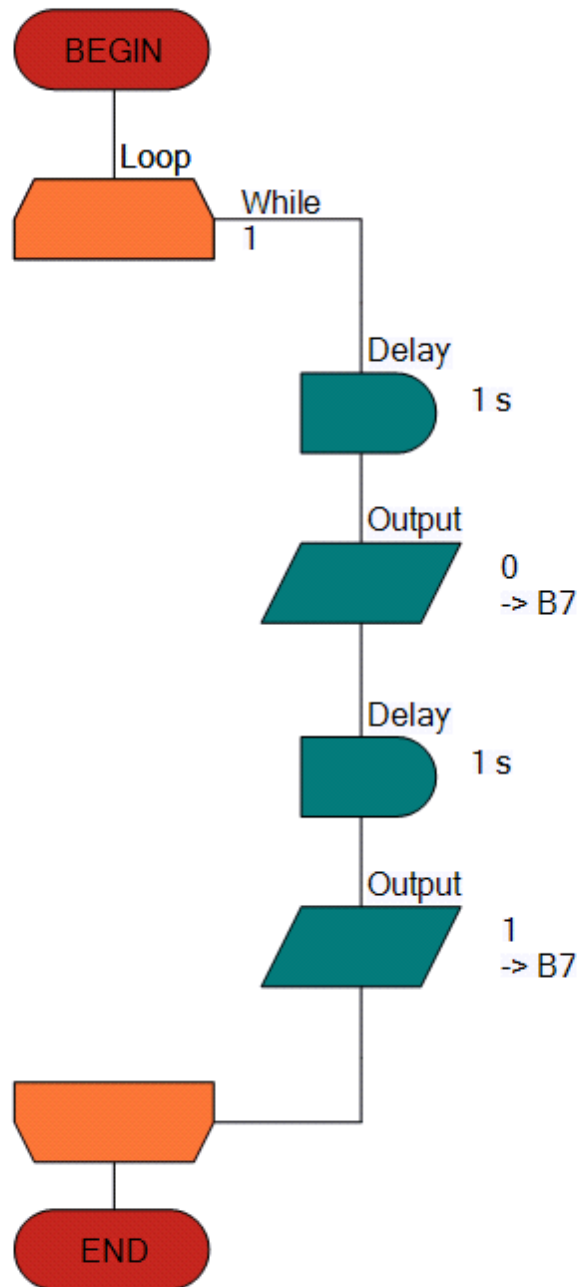
“Delay” is one of the first features we all use when learning to program. It is the microprocessor equivalent of standing in a corner, doing nothing but counting to a predetermined number, then when that number is reached, continuing with the pre-delay task - “ready or not, here I come”.



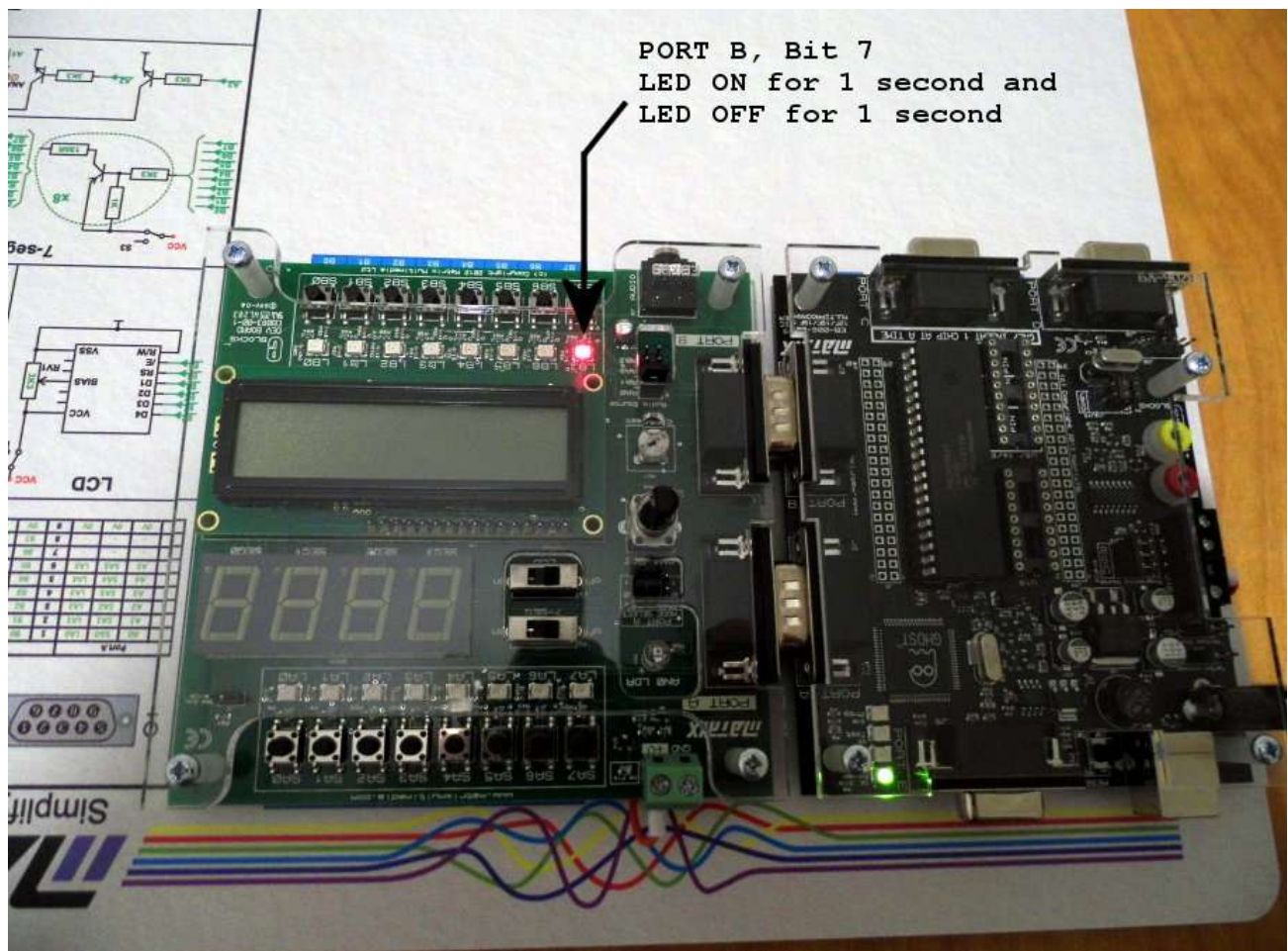
Modern microprocessors and microcontrollers perform millions of operations per second. Humans, on the other hand can not detect events at this high rate of change. We can detect audio changes of state at a maximum rate in the region of 10,000 times per second. Visually, we fare much worse, struggling to detect a flashing light at a rate of 10 times per second.

The first program we attempt when learning a new programming language, testing a new microprocessor or evaluating a new microcontroller chip is the “flash a LED at a rate of one second on and one second off in a continuous loop” program. In order to tame the speed of the microprocessor, we need a one second delay between changing the “on” to “off” or “off” to “on” state of the LED. (This is where we use “Delay”).

Once you get this working, you are well on the way to attempt more complex programs.



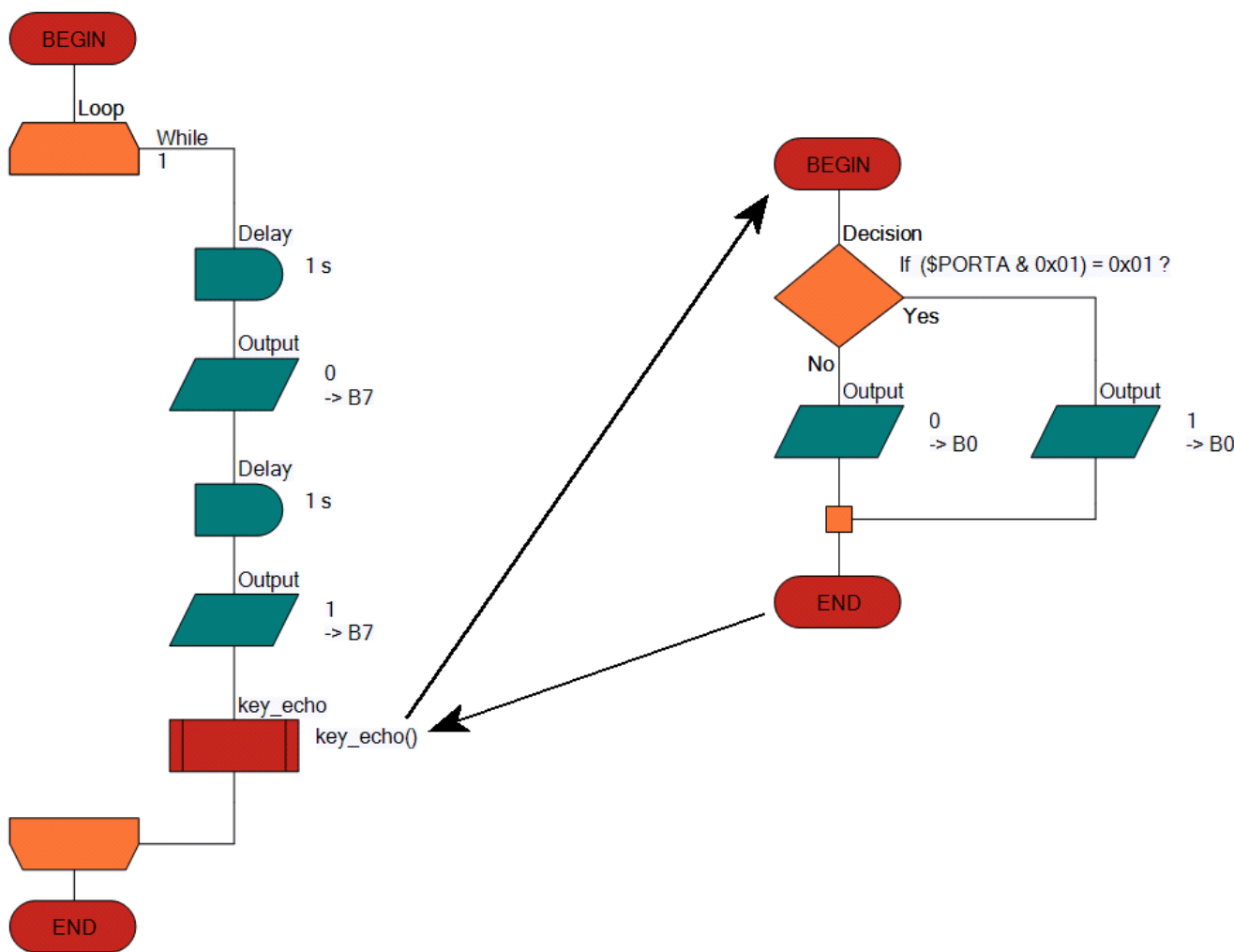
(the only way to really know if your program works is to set it free by downloading it to the actual hardware, in this case a **Matrix HP4988** )I\* development board.



...as exciting as it is, getting our first program written, built, downloaded to the Matrix HP4988 board and working perfectly. At some point we need to perform more complex operations.

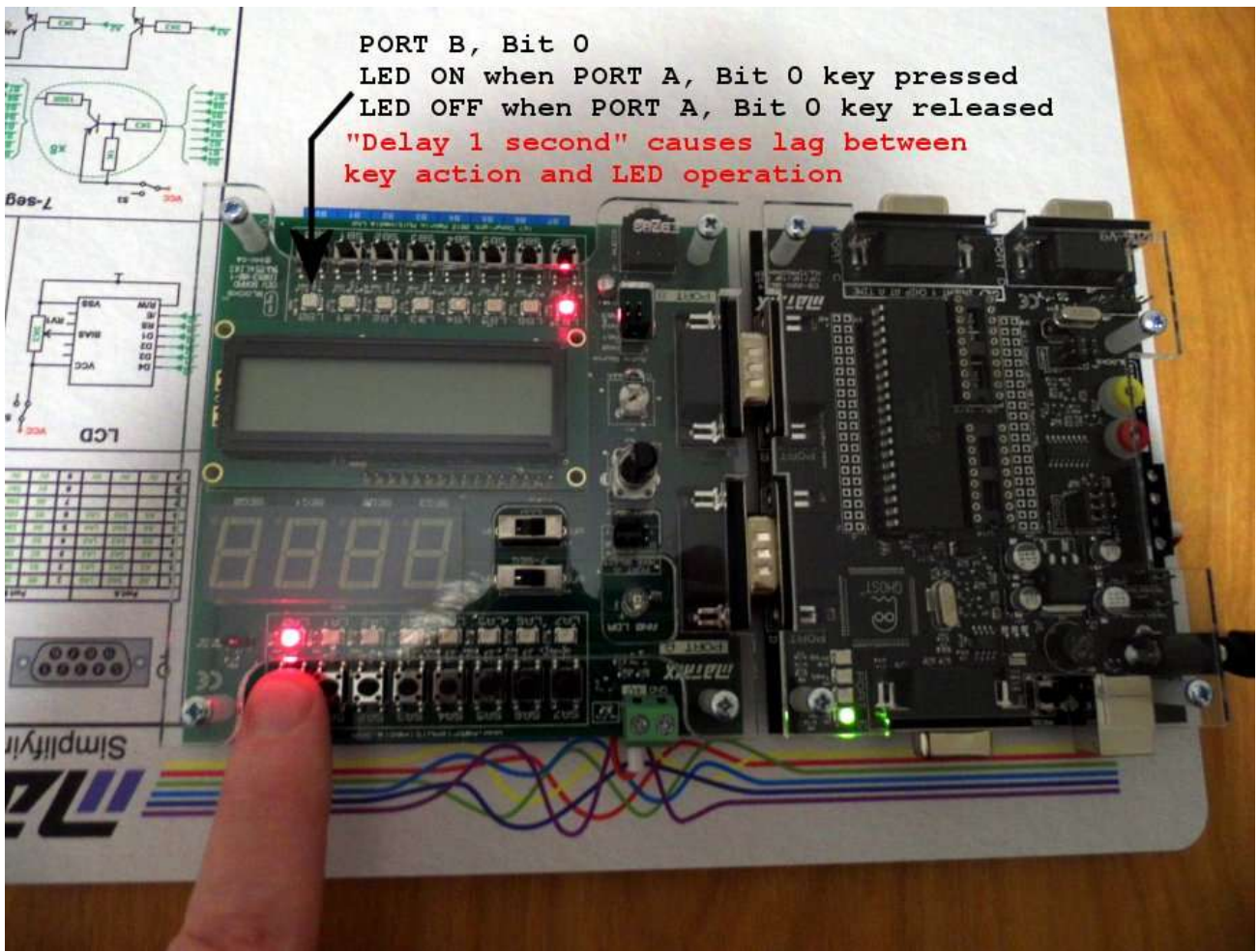
Now about flashing a LED at a rate of one flash per second, and at the same time reading a switch and echoing its state to another LED.

(here are numerous ways to achieve this task, but here is one solution4

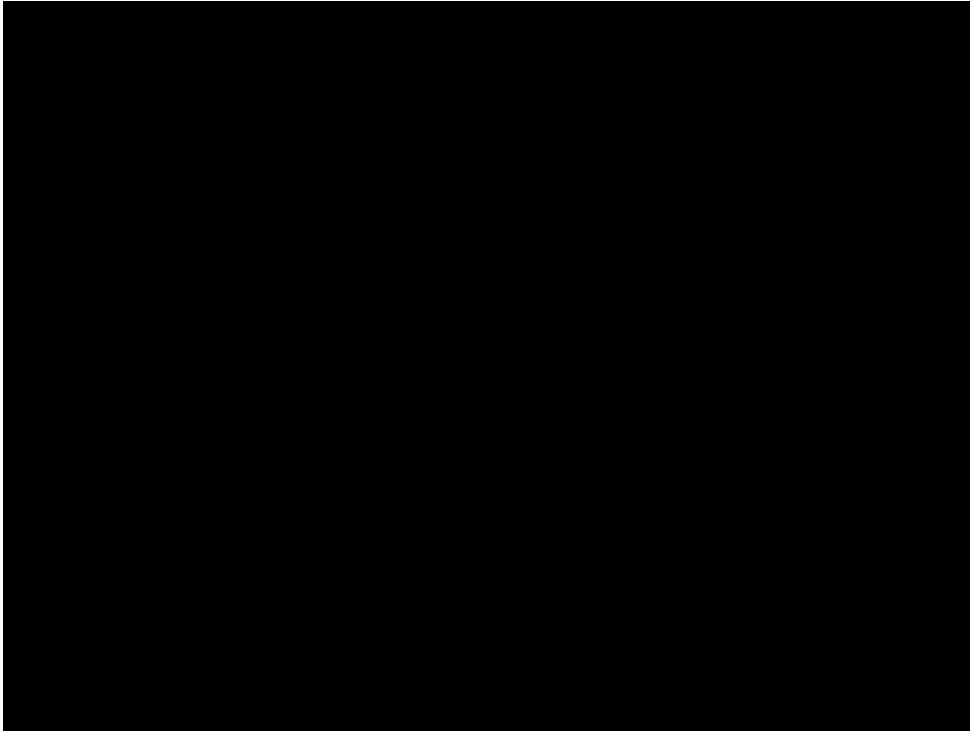


one5second5,-D5and5key5echo5)rog56.fcf#

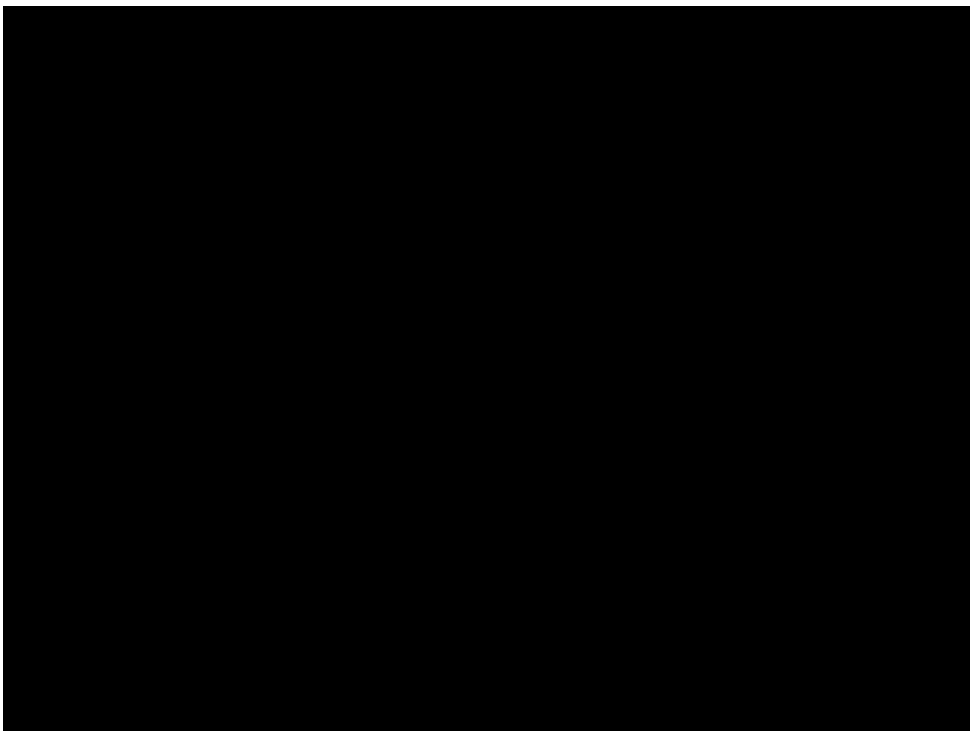
. simple solution like this suffers from an obvious problem when you run it on the atri# !)' /00 board. +ecause the program does nothing during a "Delay" and there are two one second delays between running the "testing for a key press and turning on the relevant , -D" macro, this lag is noticeable.



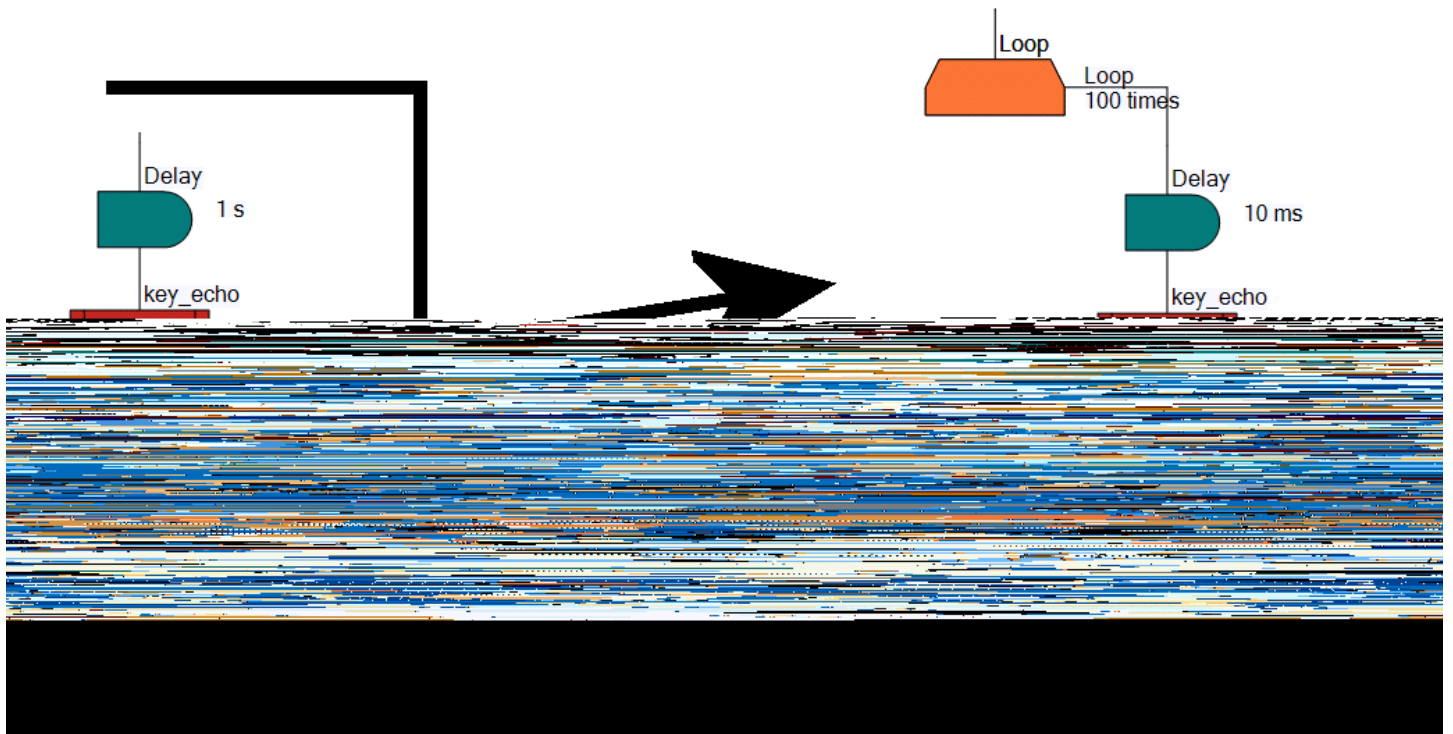
-ventually, the program catches up and turns on the , -D.



(he same lag is also noticeable when releasing the key and the , -D  
e#tinguishing.



"e can improve the situation by splitting the one second delay into a number of smaller delays 7say 6%% # 6%ms8 and running the key5echo78 macro after every small delay.

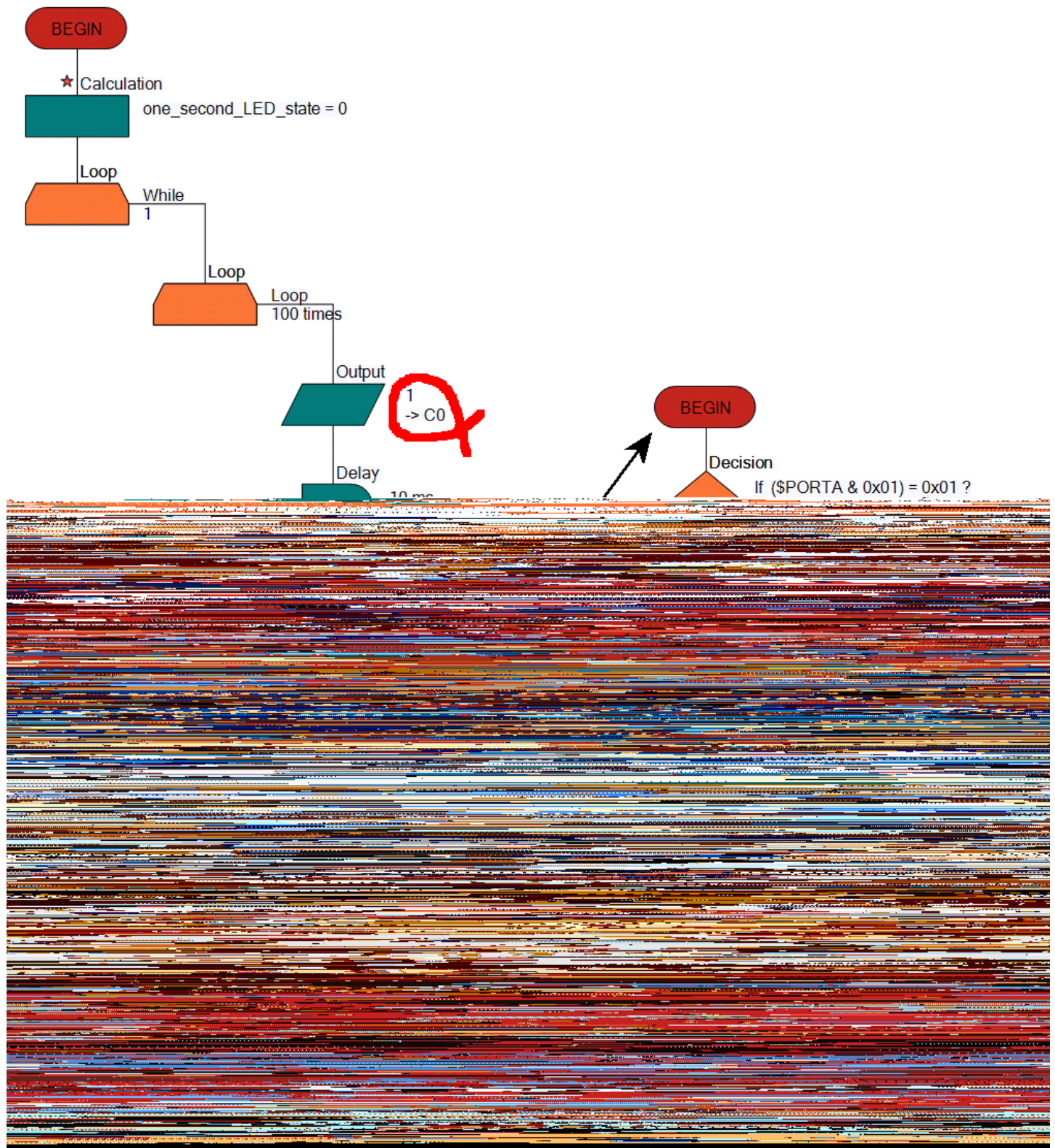


If you download this solution to the atri# !)' /00 board, you can see the , -D flashing on and off at a rate of one flash every second and pressing the key immediately illuminates the , -D, while releasing the key e#tinguishes the , -D immediately.

!owever, as we noted earlier human reactions are not fast enough to really appreciate what is happening.



(his modified solution sets )&9( \*, bit % to :; <ust before calling the 6%ms Delay and clears )&9( \*, bit % to %; when the 6%ms Delay has finished. If we connect an oscilloscope to this port pin we can “see” what’s happening during a delay and between delays.

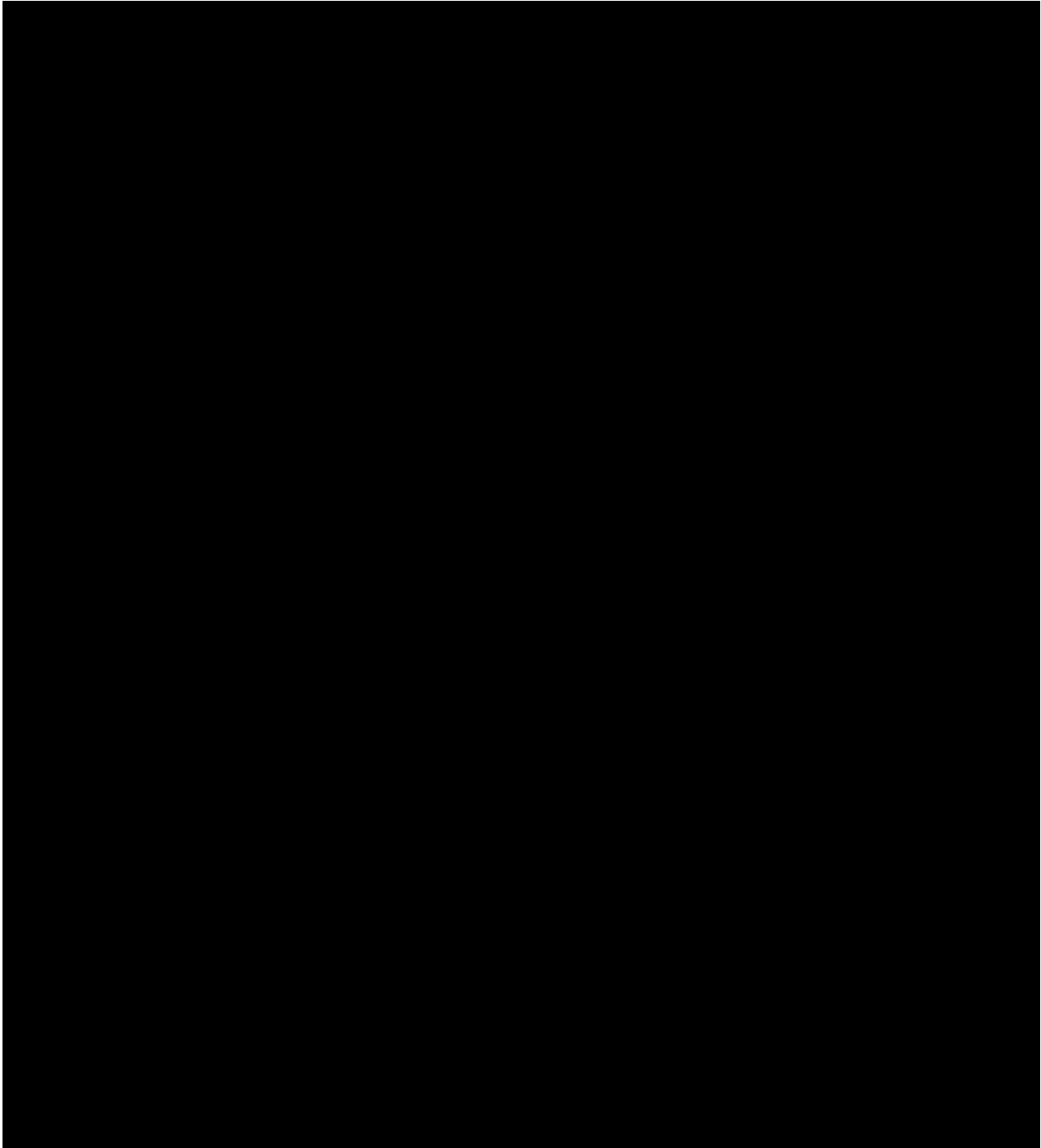


one5second5, -D5and5key5echo5) rog5\$. fcf#

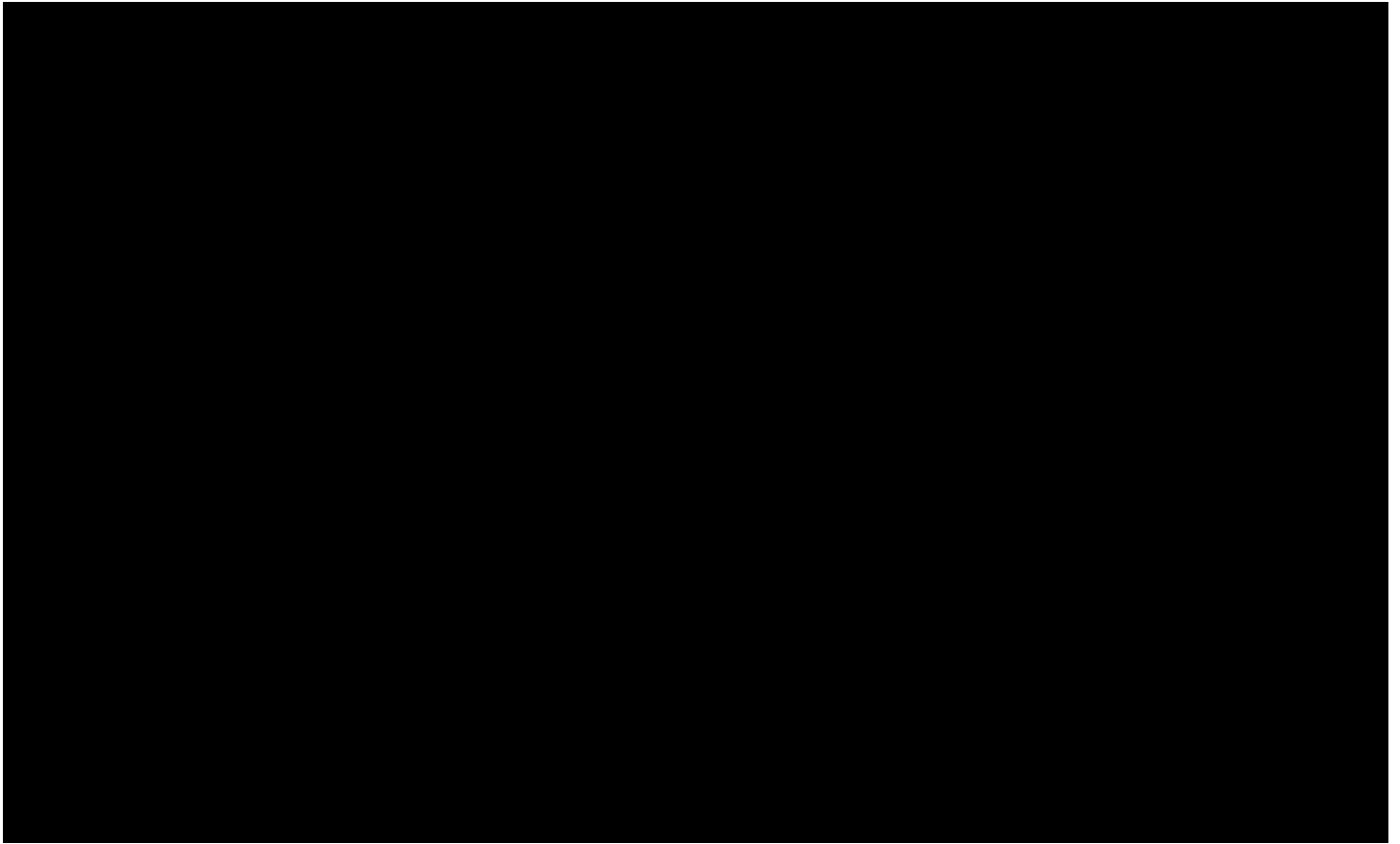
=ote4 (he atri# !)' /00 board has a ghost feature that we could use to do this measurement. .gain there are many solutions to any problem.



(he )&9( \*% pin seems to stay at :; for a very long time, indicating that the program spends most of its time in Delay.



Looking at the timing waveforms in more detail, we can see how much time the PIC chip on the PIC16C610 board spends in "Delay" and how much time running the rest of the program.



$$\text{Non-Delay to Delay Ratio} = \frac{17.20\mu s}{10ms} \times 100 = 0.172\%$$

So the microcontroller spends over 1% of the time running Delay and less than 0.2% of the time running the rest of the program.

This very simple example illustrates a very real bear trap when using "Delay" to time an event by counting the number of delays from the start of an event (e.g. a key press) and counting the number of delays until a predetermined count is reached.

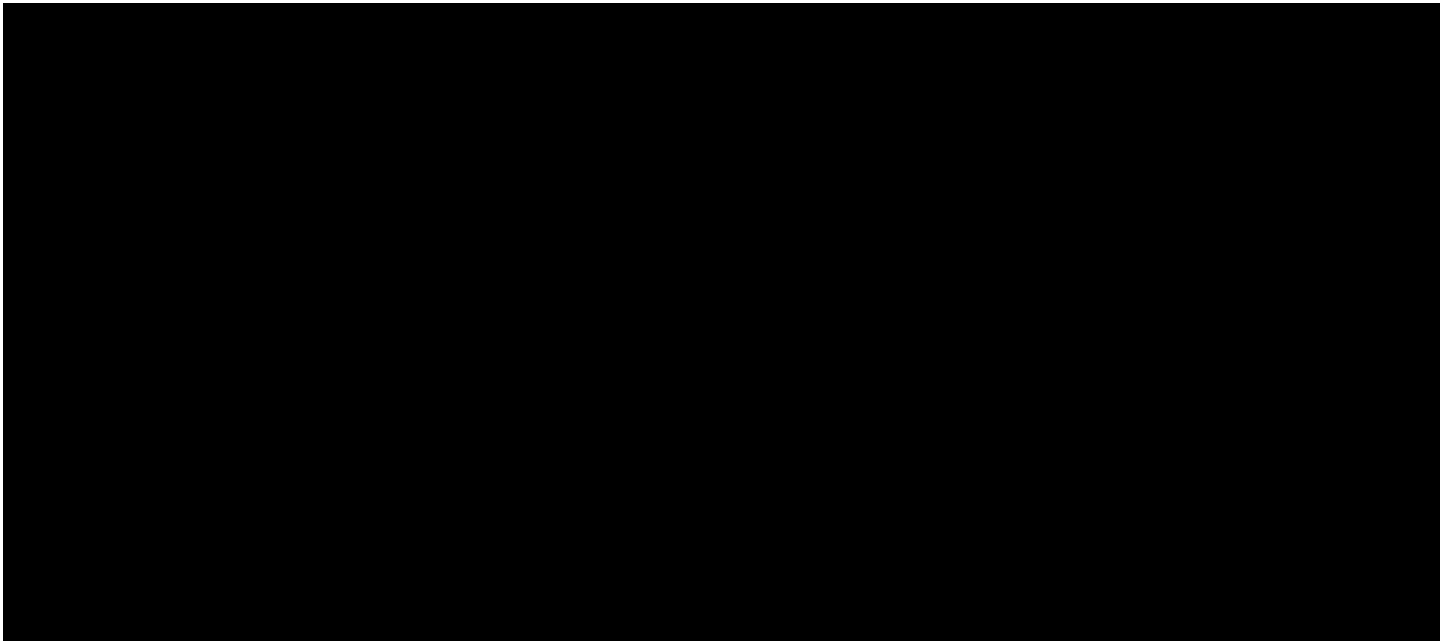
Consider a situation where you have a more complicated program with multiple routes through your program, based on the outcome of decisions, number of loops run, etc. (The time between delays will not be a constant value. You could estimate an average time and calibrate your system accordingly.

Assume you do this and you are accurate to 0.2% which sounds fine. Now consider the case where you need to measure the energy output of a solar farm over a one year period. By the time you get to the end of the year with an accuracy of 0.2%, your timing, you could be out by more than 18 days.

!

As we have seen, Delay is a “killing time” function. (That can be good to try out an idea quickly, get to grips with a new microcontroller chip and many other uses. Delay is a waste of time, but that does not mean that it is useless.

Even the most basic microcontrollers have at least one hardware timer that once setup, runs independently of the main program. (Think of it like an egg timer. Rather than standing in the corner counting numbers to kill time, set the egg timer running and do something else until the sand runs out, then carry on with whatever happens after the “ready or not” part of your program.



!

"

#

Microcontroller timers run on interrupts. Basically your main program runs until the timer causes an interrupt. When the interrupt occurs, a special macro you have written for the timer interrupt is triggered. When this special macro comes to an end, the microcontroller continues running your main program from where it was before the interruption.

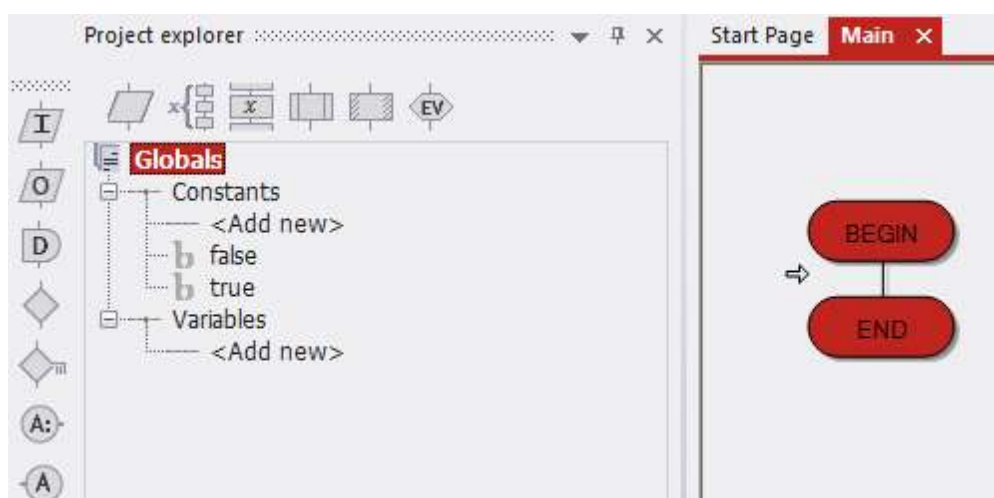
(here is a trade off between how fine you want your time increments, how much interruptions you can tolerate in your main program and the maximum time interval you expect to measure.

Generally we choose 6ms as the basic system timer tick, meaning that it is not too intrusive to your main program. Also, if you use an unsigned integer range 0 to 65535 to store the number of 6ms ticks, you can measure up to just over 60 minutes with a 6ms resolution very easily.

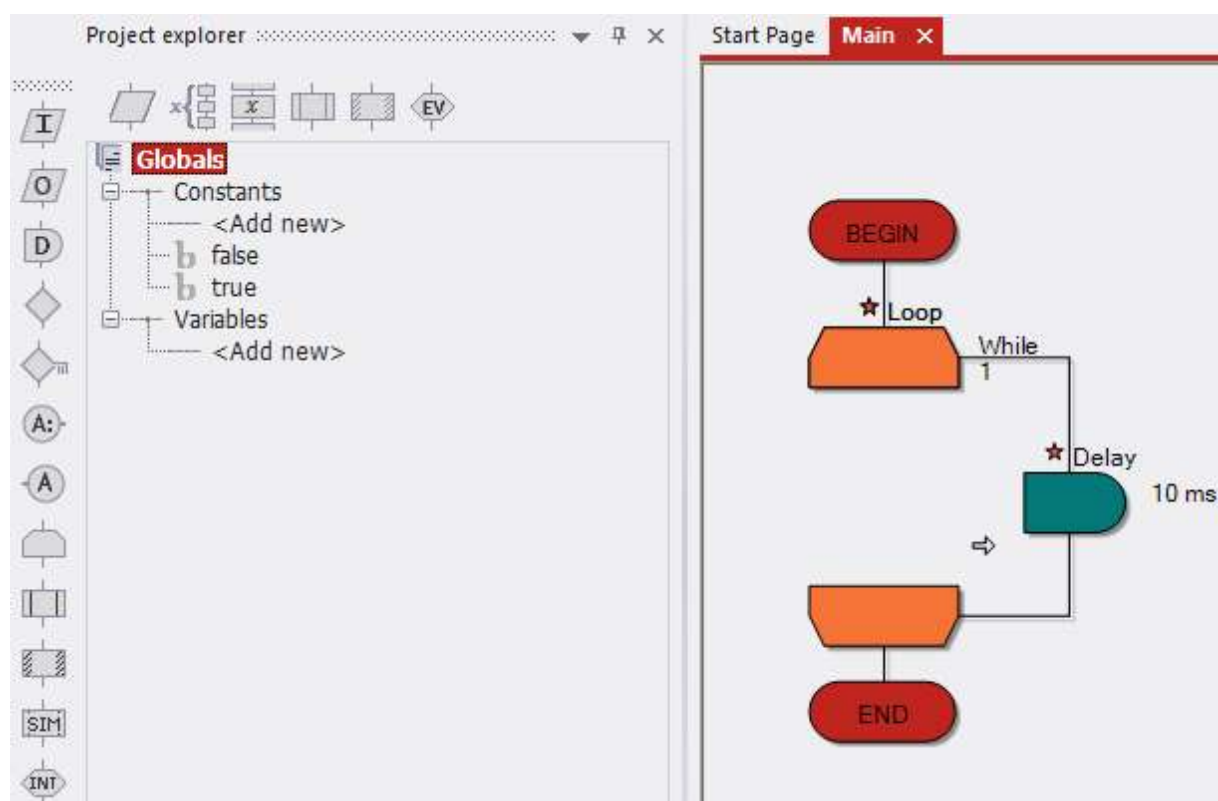
(the other point to note is that we want to spend as little time as possible running the program inside the timer interrupt macro. (therefore, we do the bare minimum of processing inside the timer interrupt macro.

\$ % & " #  
 >etting up a microrontroller2s timer required a lot of steps but the  
 Elowcode F software makes it much easier, even if it seems daunting at  
 first.

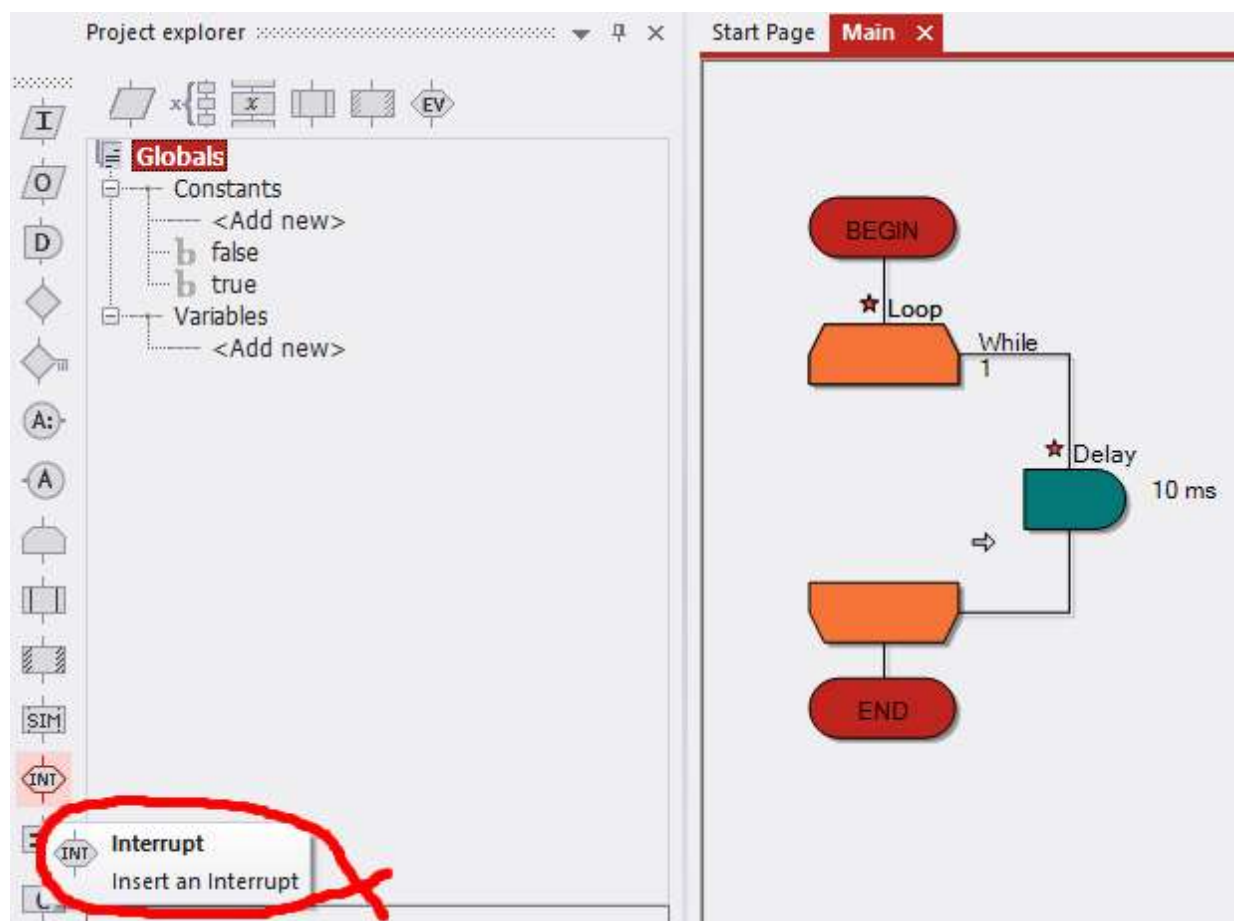
768 \*reate a new pro<ect4



7\$8 For your main program, setup a “forever loop” with a delay.

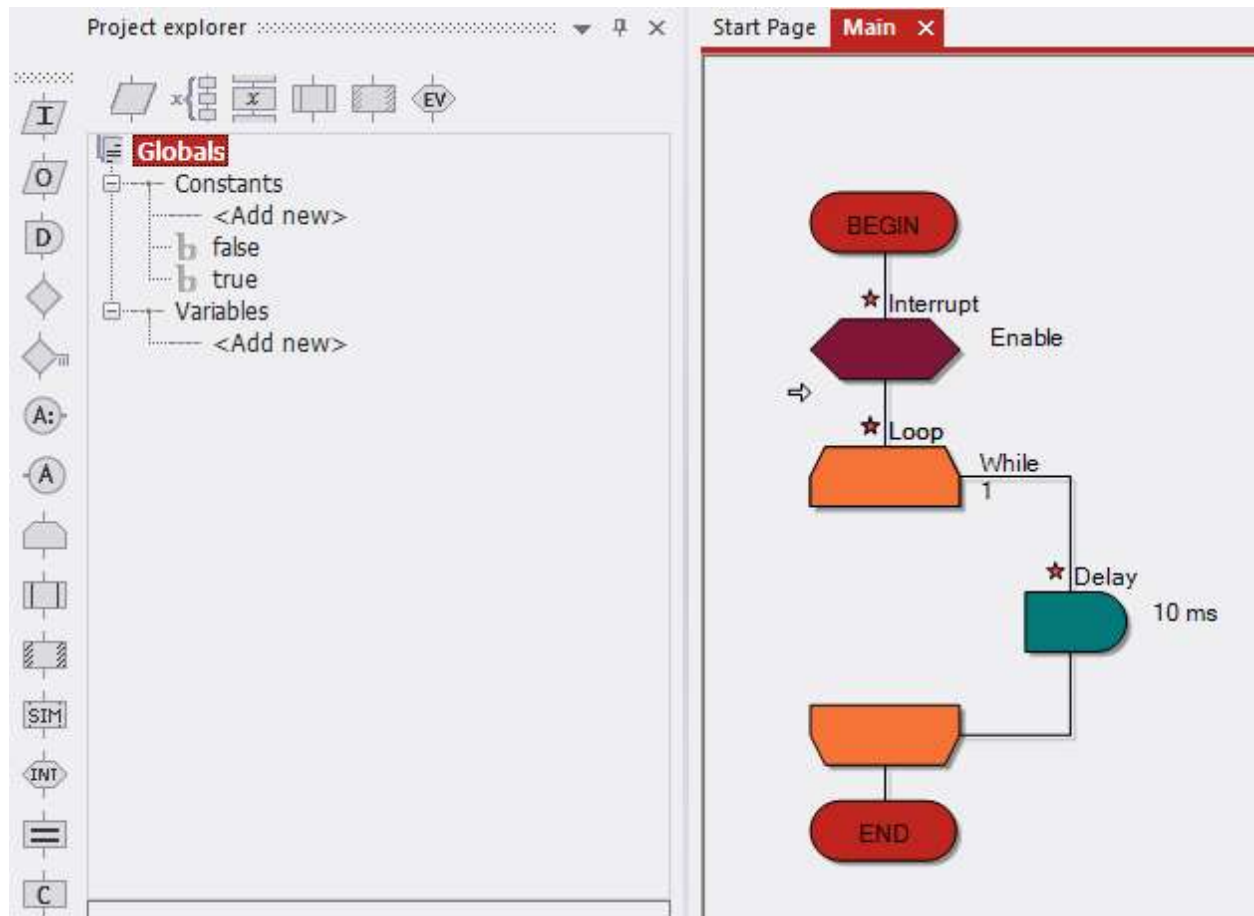


7A8 >elect the Interrupt icon

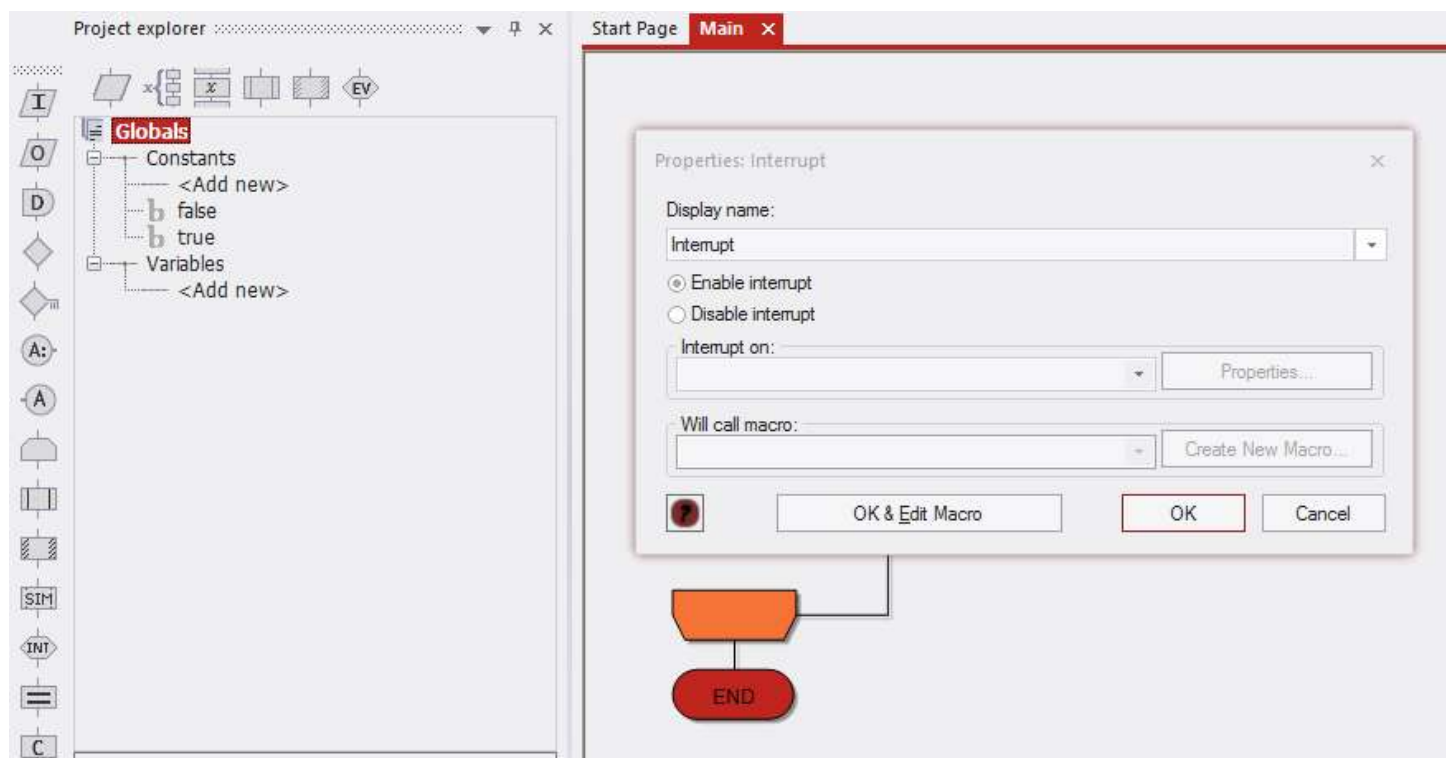




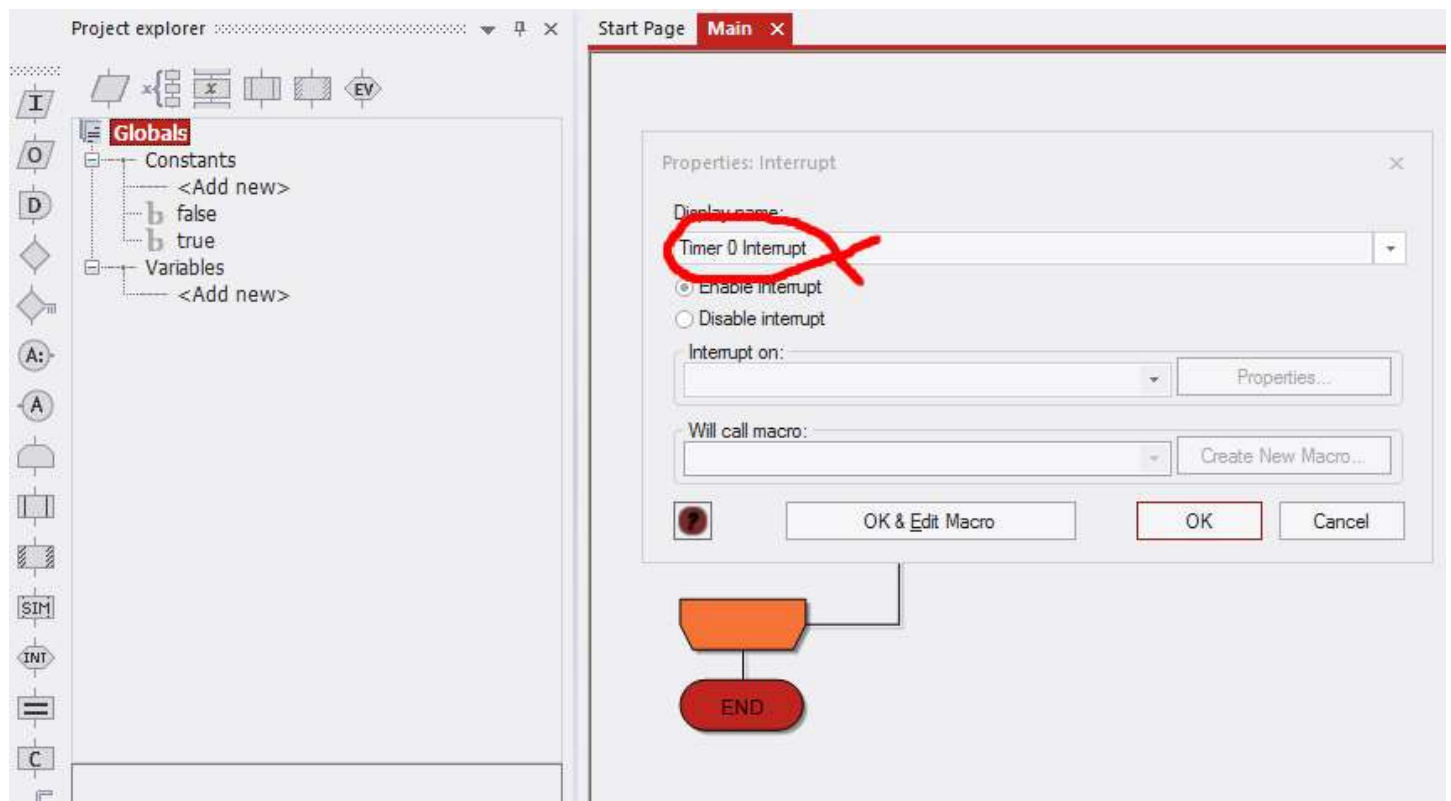
7'8 )lace the Interrupt at the beginning of your main program before the start of the “forever loop”. (his is really the Interrupt setup function and only needs to run once at the start of your main program 7i.e. before the “forever loop”8.



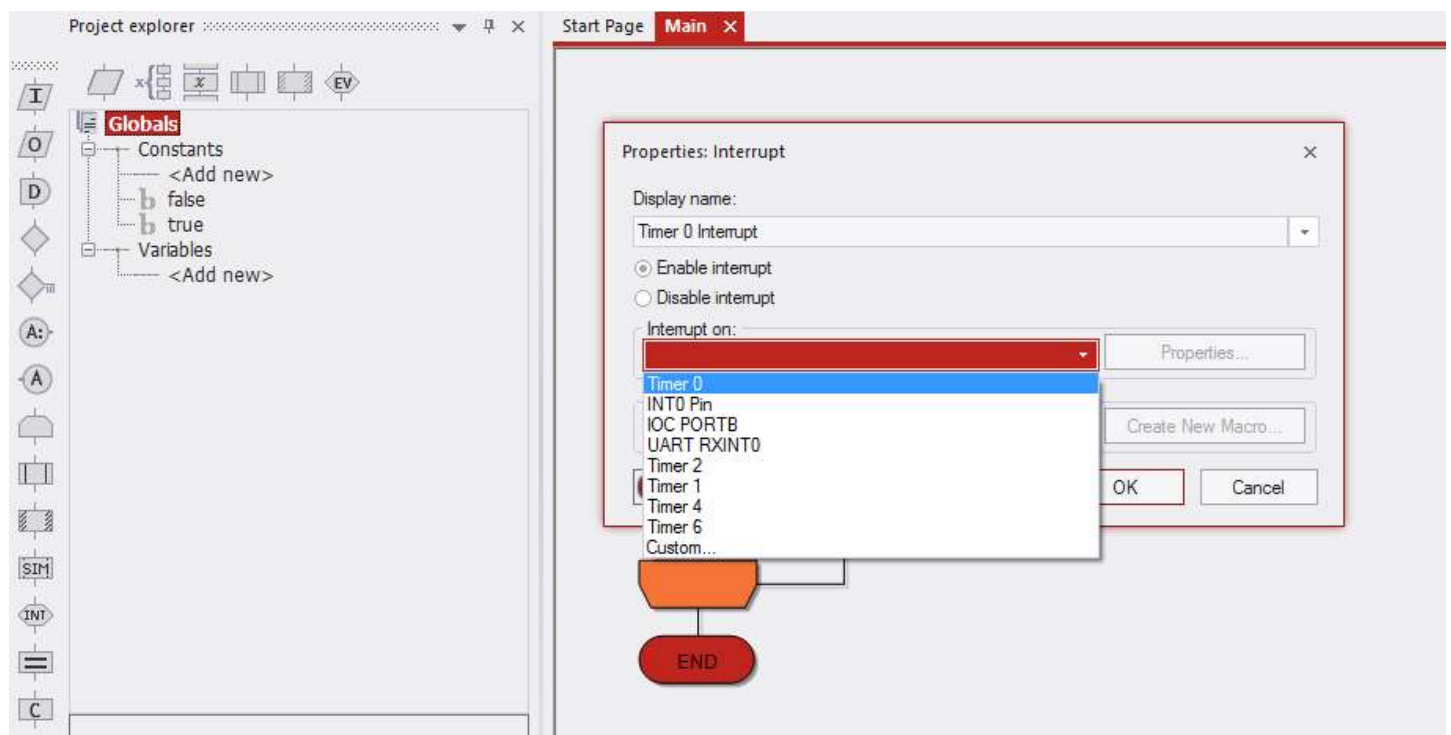
7:8 =ow you need to configure the timer and how you want it to work.  
>elect the Interrupt ob<ect and open the properties.



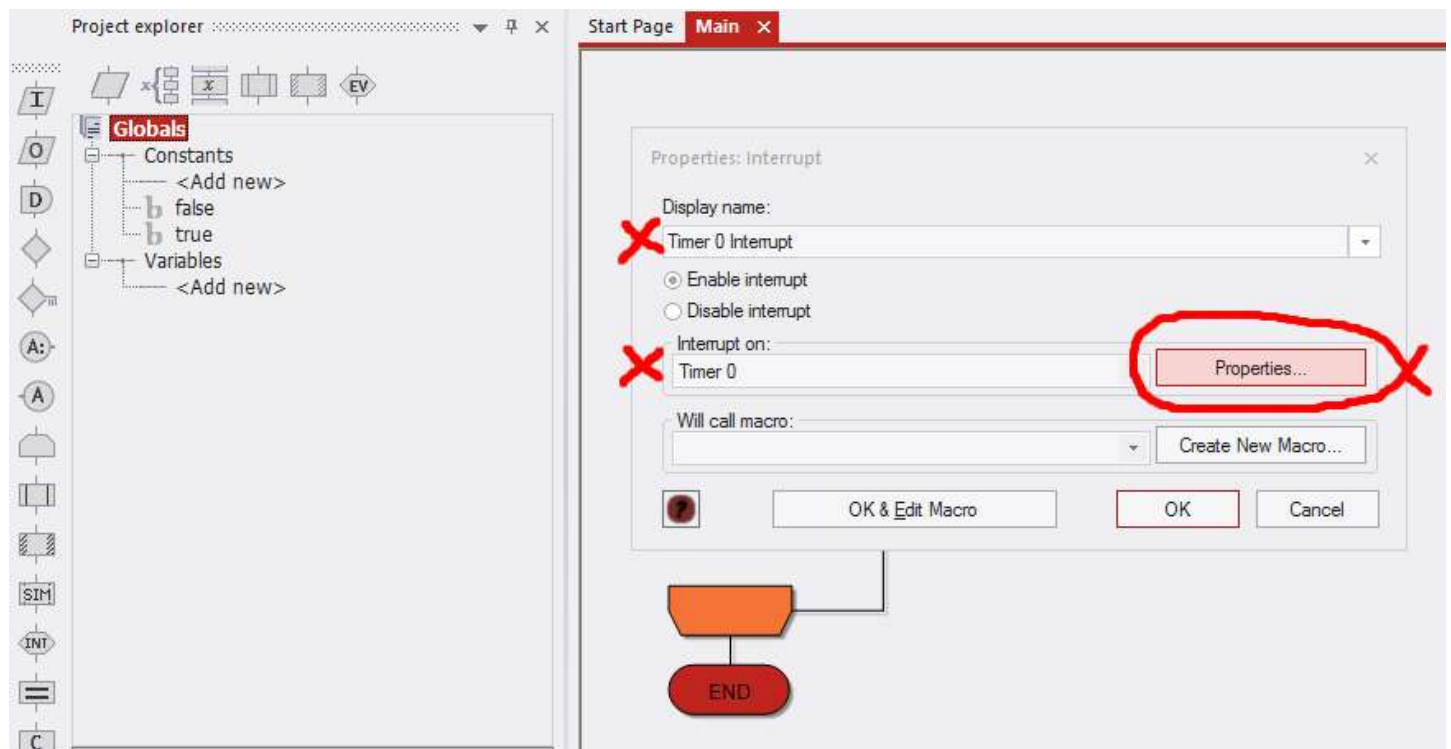
7D8 Give the Interrupt an appropriate Display name like “(imer % Interrupt”.



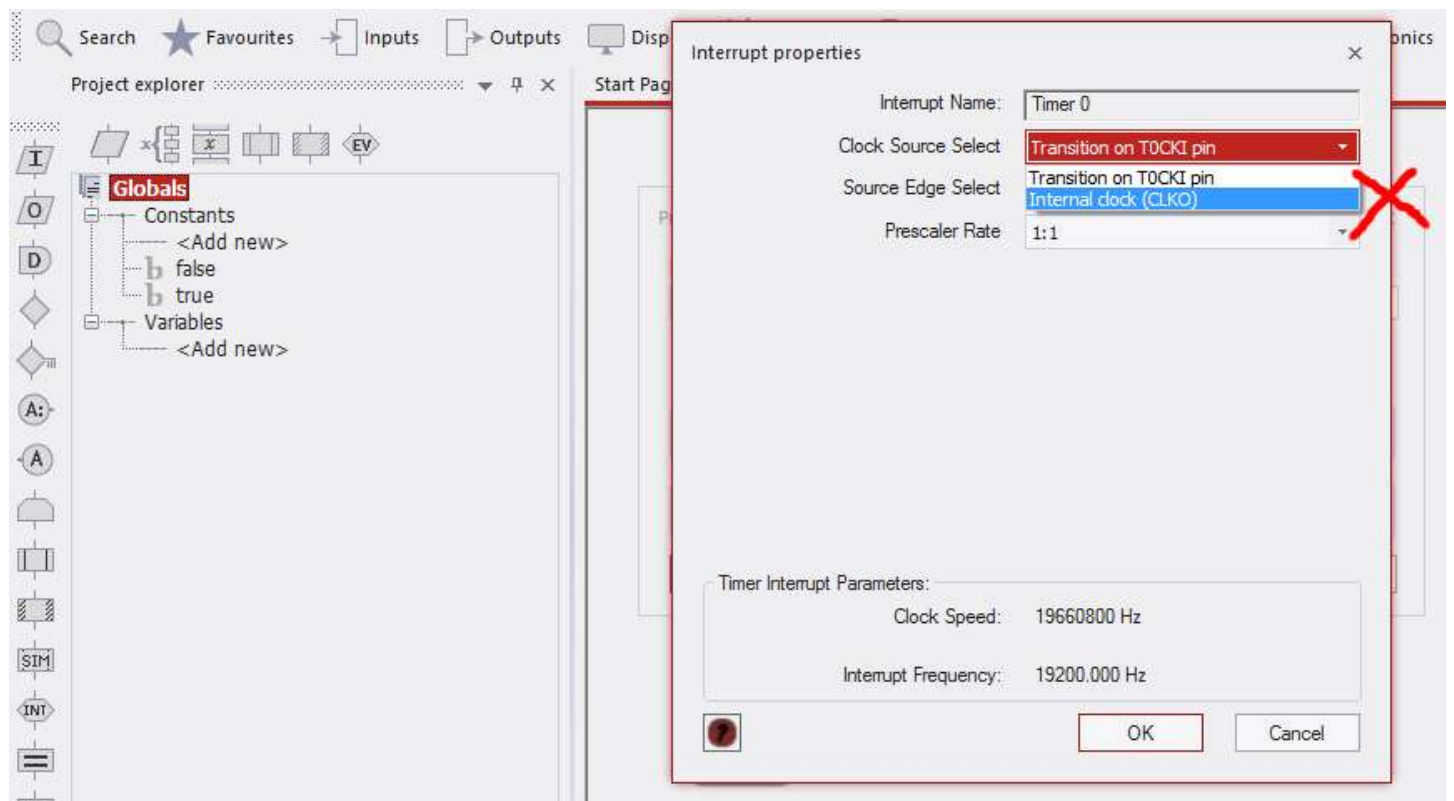
7F8 @ou can setup various events to cause an interrupt. "e need to use (imer % to cause an interrupt for us.



708 Now that we have tapped into the (imer % interrupt, we need to tell the Flowcode F how we want to use it.

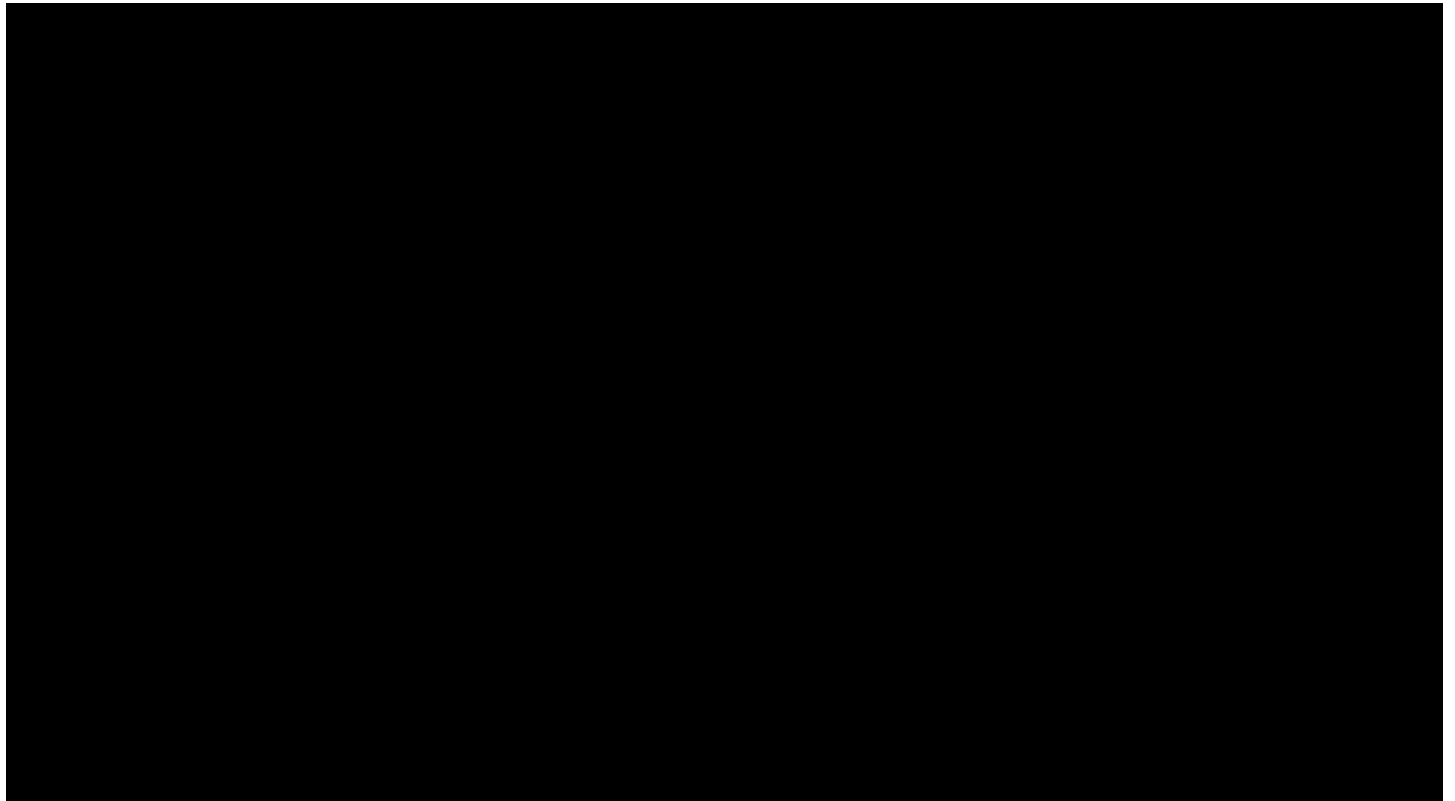


7/8 (he timer can derive its clock from pulses on an input pin of the microcontroller or from the internal clock of the microcontroller. We need to choose the internal clock of the microcontroller 76/00%G in the case of the atri# !)'00 board8.

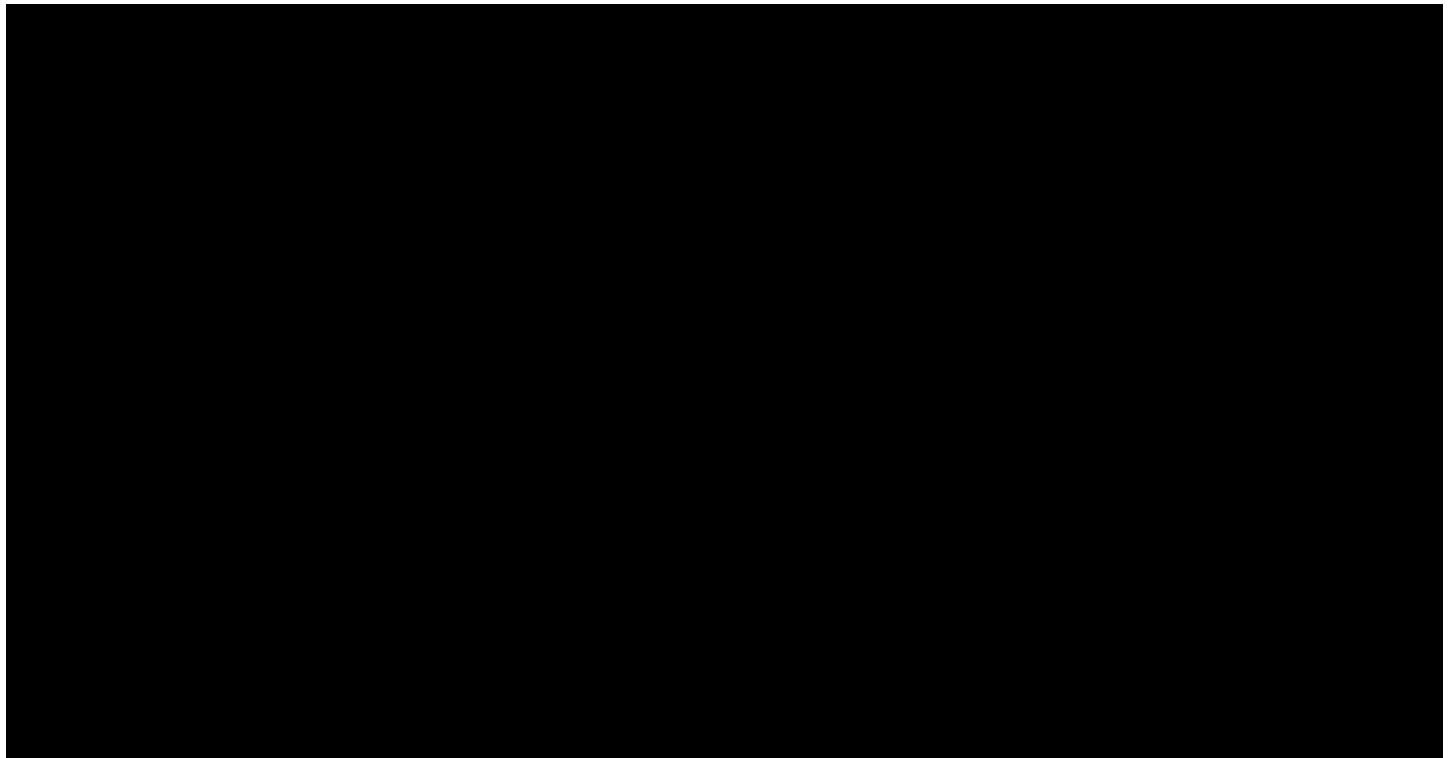




The microcontroller has a prescaler between the 60MHz clock and the timer clock input. This works like a reduction gearbox, slowing the clock speed for the timer. We need a reduction of 64, giving us an interrupt rate of 300Hz. This leads to an interrupt being generated every

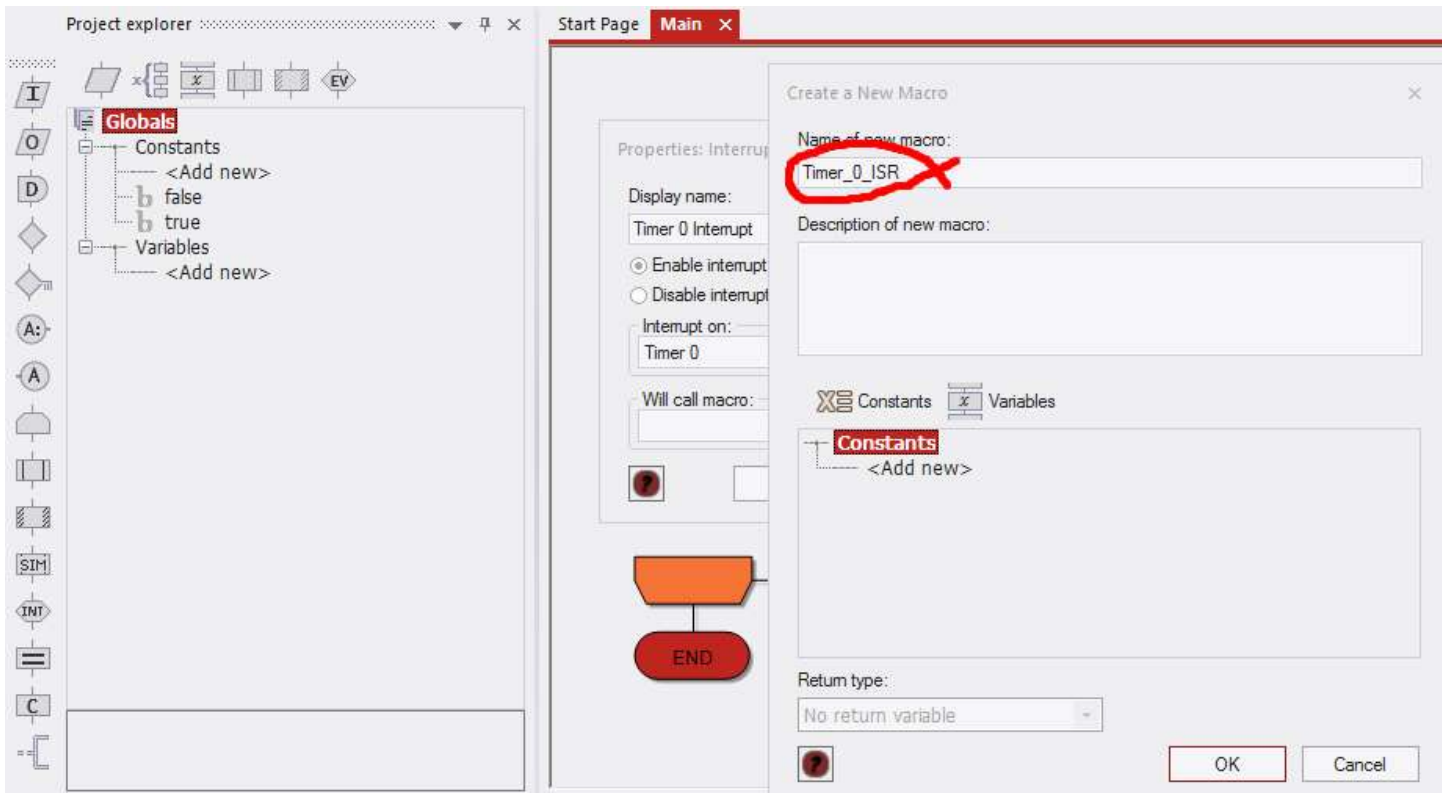
$$\frac{1}{300 \text{ Hz}} = 3.33 \text{ ms}$$


7668 Now that we have setup (imer % to interrupt every  $3.33ms$  , we need to create the macro that is called when the (imer % interrupt occurs.

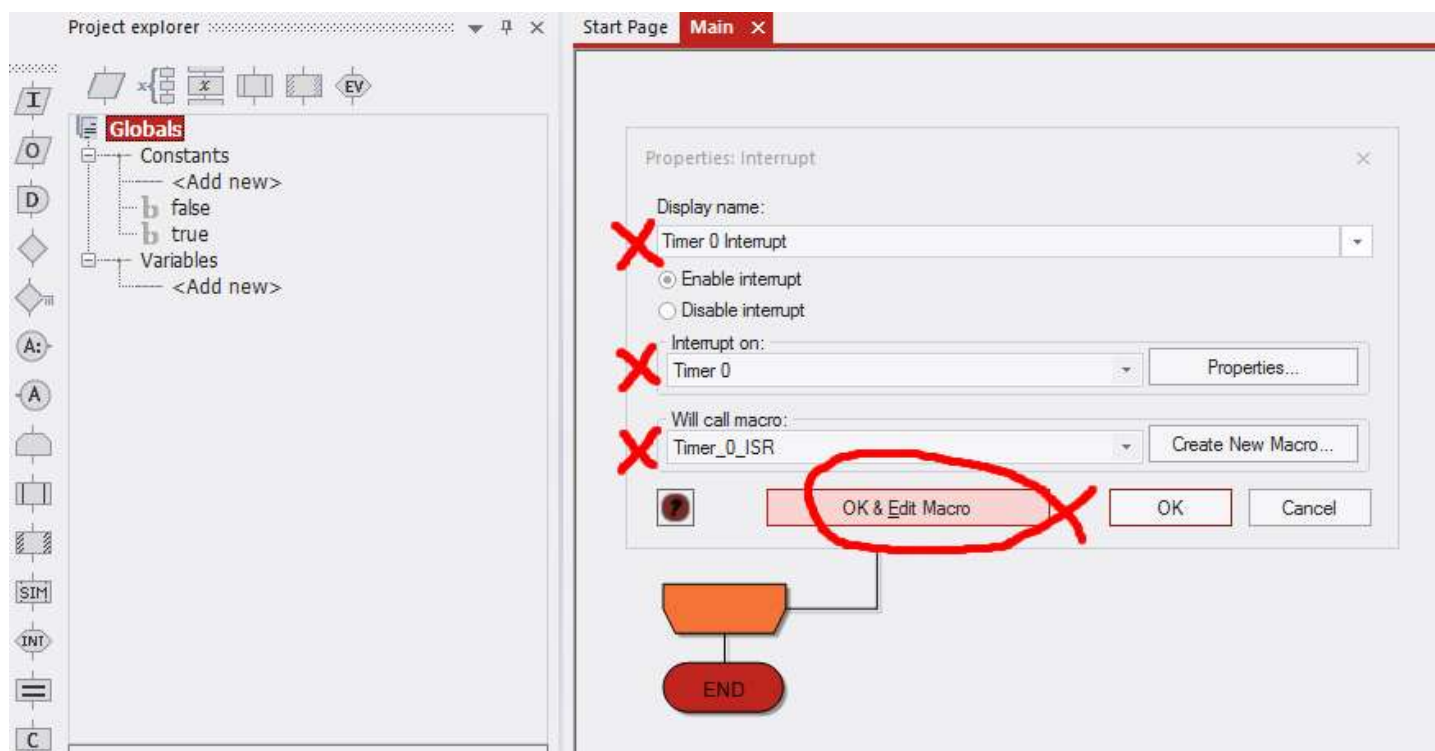


76\$8 "e will call this special macro "(imer5%5l>9". l>9 is an abbreviation for "Interrupt >ervice 9outine", so we know that2s not a normal macro.

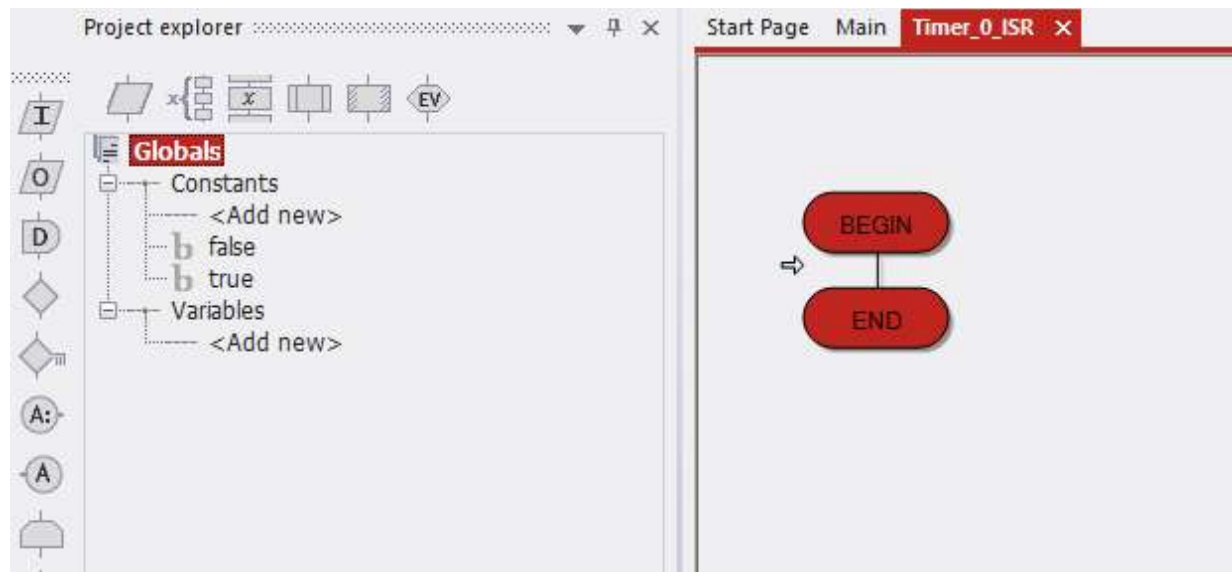
. normal macro is called from somewhere in your program 1 i.e. you decode where it is called. .n l>9 macro is called asynchronously - in this case by the (imer % hardware and can occur anywhere in your normal running program. "hen a normal macro finishes, your program flow returns to your where it was called from. "hen an l>9 macro finishes, the microprocessor continues running your program from wherever it was before the l>9 macro interruption.



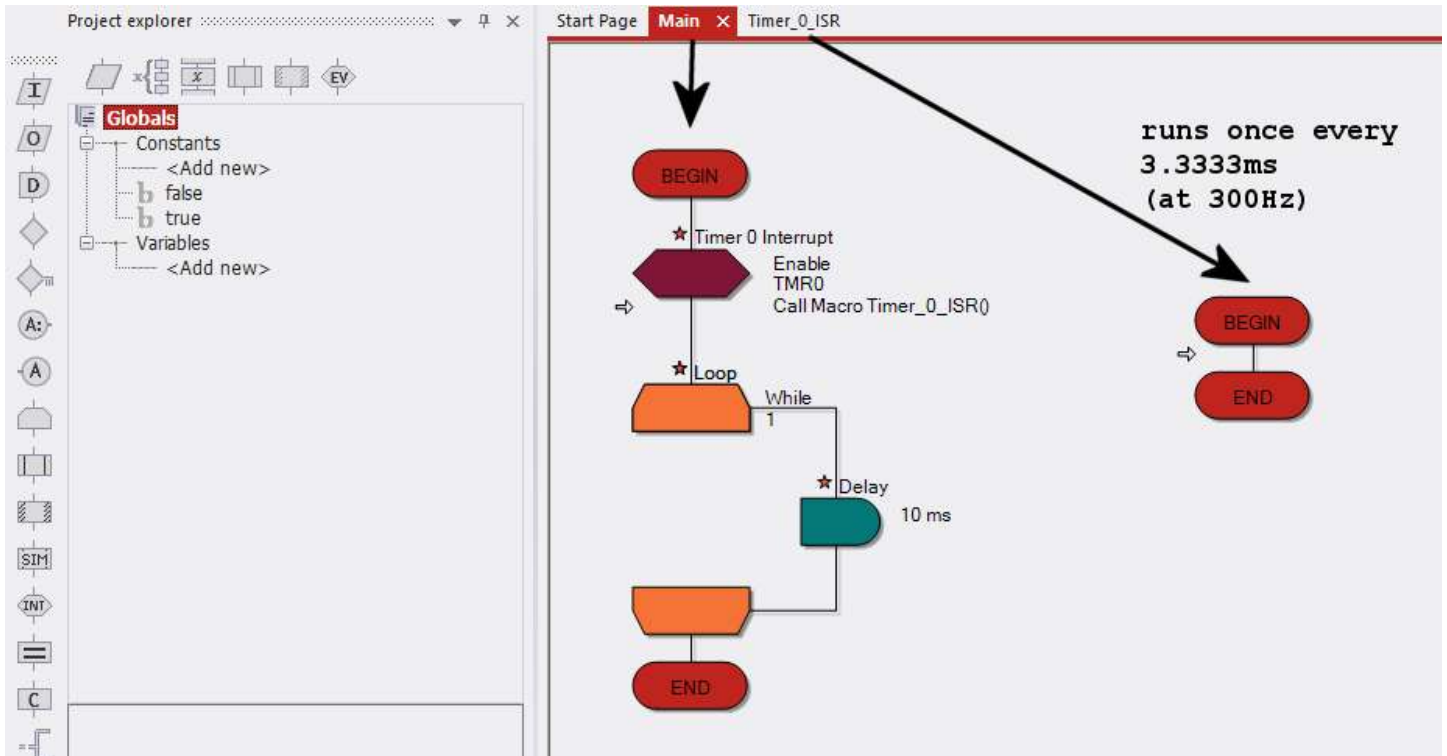
76A8 =ow we can create and edit the (imer5%5l>9 macro.



76'8 "e now have an empty macro called "(imer5%5I>9" that is called when a (imer % interrupt occurs 7every 3.33ms 8.



76:8 (this gives us a skeleton of a main program and an interrupt service routine that runs every 3.33ms).



Why did we choose a 3.33ms for the timer interrupt rate?

Because it is the slowest clock rate we can use to get 6ms easily.

$$3 \times \frac{1}{300 \text{ Hz}} = \frac{1}{100 \text{ Hz}} = 10 \text{ ms}$$

and we can count a whole number of 3.33ms pulses in software to get 6ms.

If we used a prescaler of 64, we would need to count 6.6 pulses and we can only count whole number of pulses.

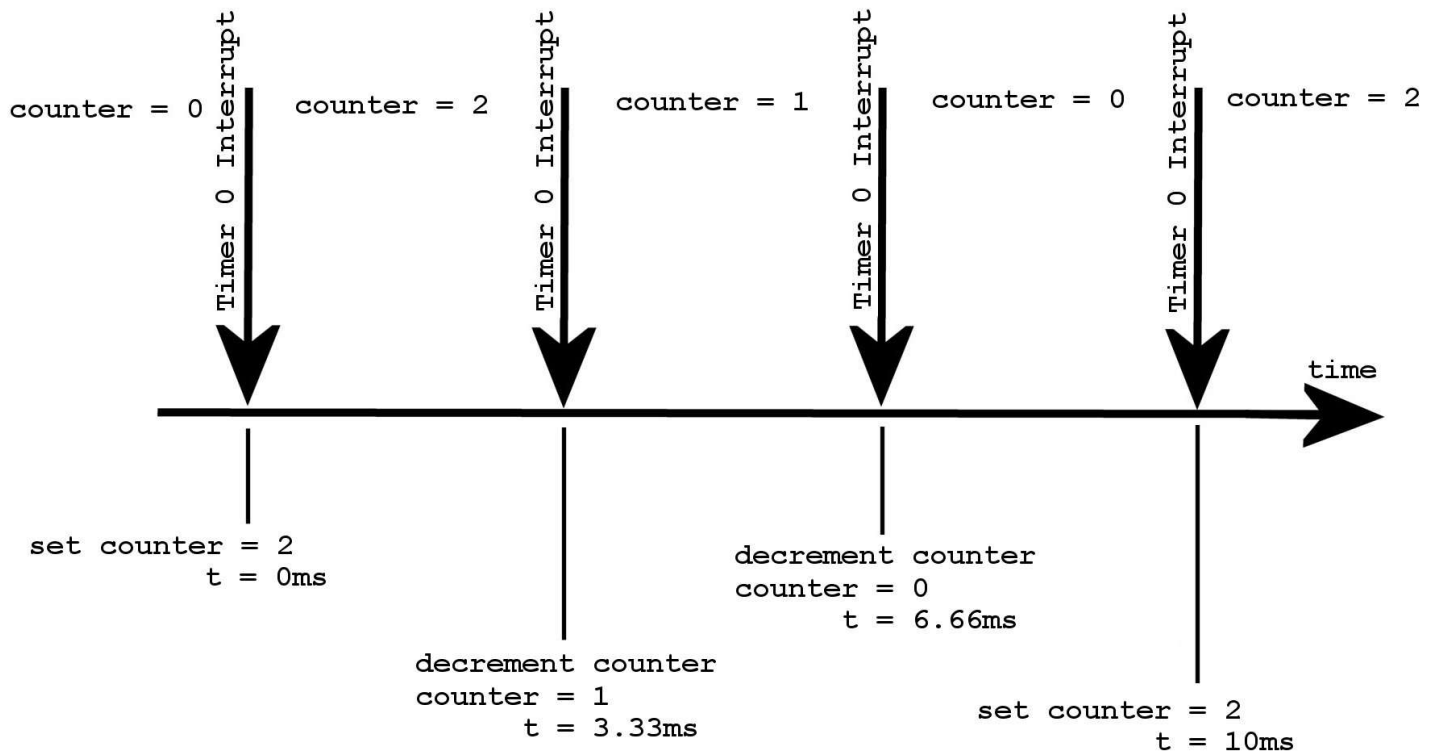
If we used a prescaler of 64, we would need to count 12 pulses, which works, but the timer interrupts occur every 1.66ms which is twice as often as when we use a 3.33ms.

(Therefore, a rate of 3.33ms gives us the lowest interrupt rate that can yield exactly 6ms increments of time for our system clock.)



' & ( &

\*ounting A times  $3.33ms$  shouldn't cause a problem. (here is another bear trap that catches us out when counting on microcontrollers. Normally, humans would count down A, 5, 6 done. However, it is more efficient for microcontrollers to count down to zero, because they can easily and sometimes automatically check for a zero result. So, in the world of microcontrollers, we normally count down 5, 6, 7 to get  $3 \times 3.33ms = 10ms$ .



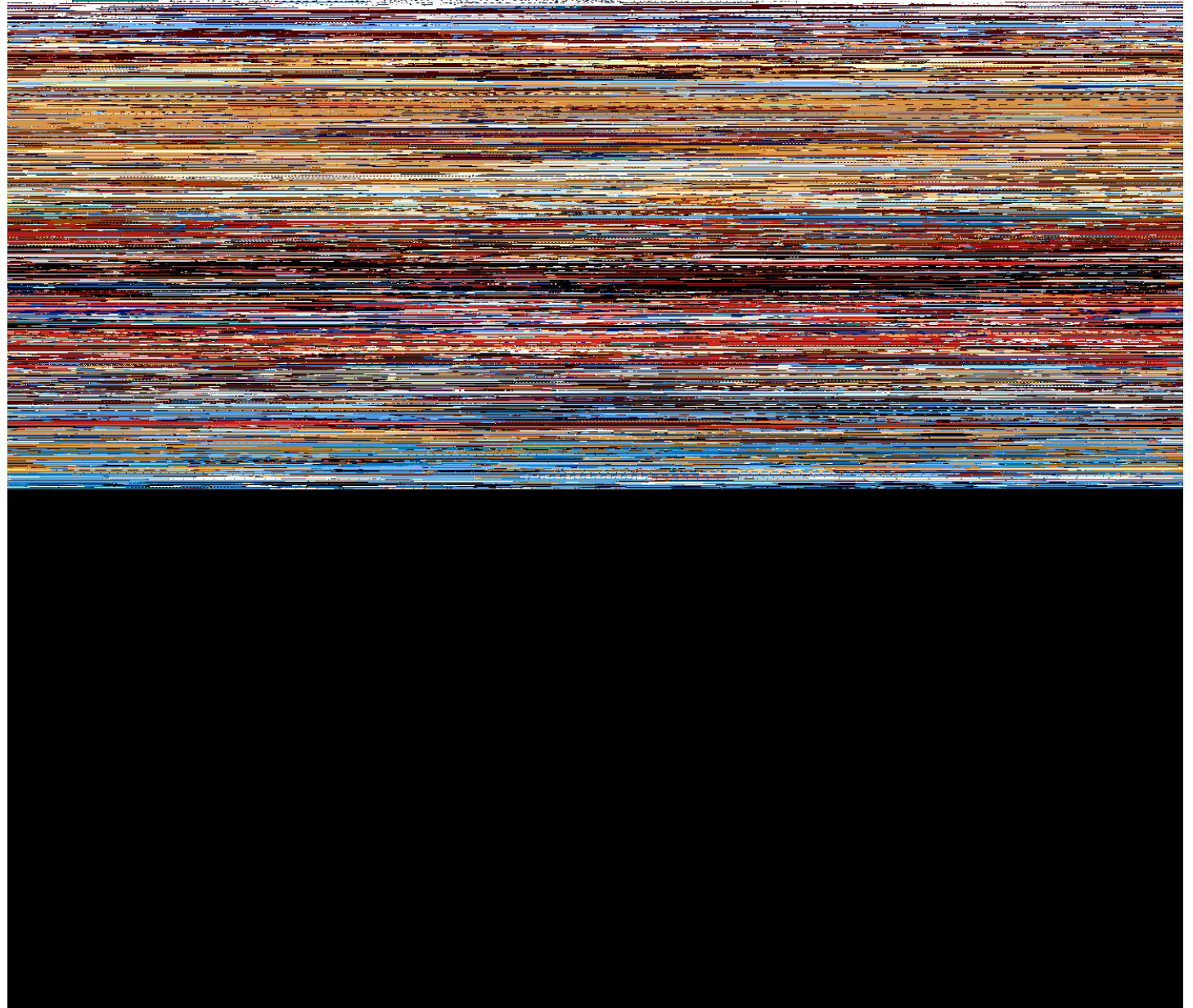
(his program is derived from our skeleton above and does nothing in the main program but toggles the state of )ort \*, +it % every time we count  $3 \times 3.33ms = 10ms$  .

**main**

BEGIN

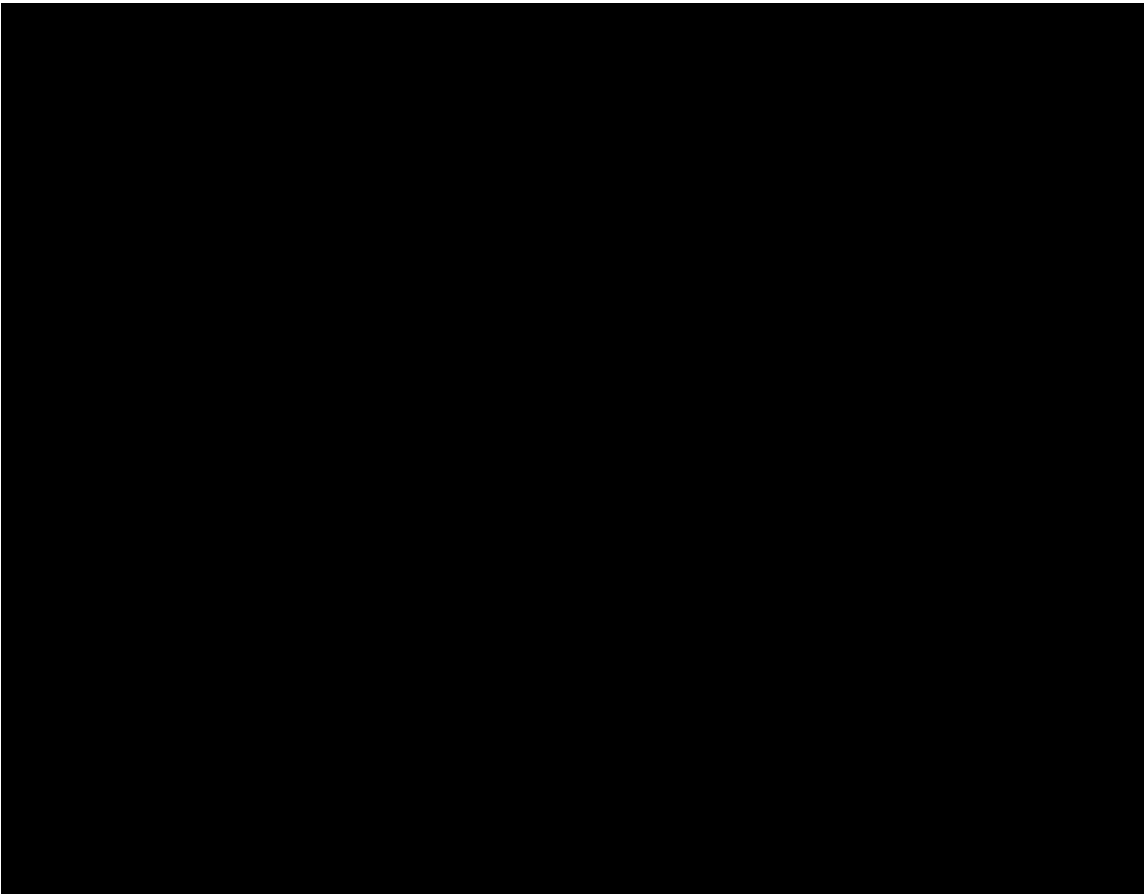
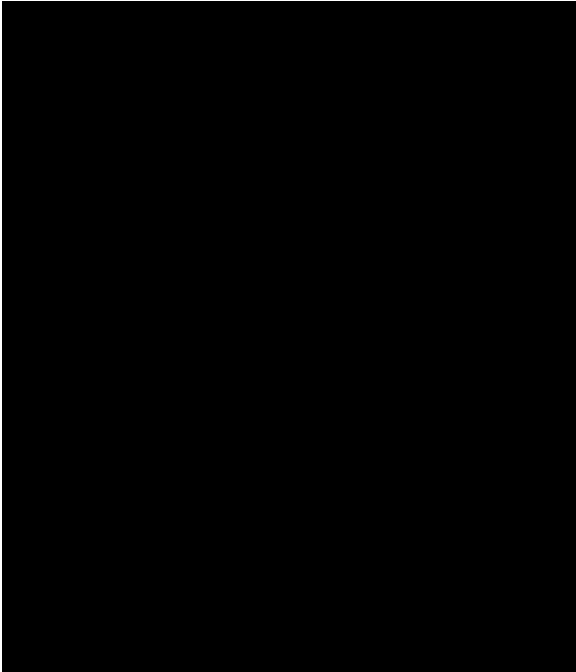
Initialise count values

Time 0.4ms, millisecond counter, COUNT FOR TEN MILLISECONDS



(imer56%ms5tick.fcf#

When we connect port \*, +it % to an oscilloscope and measure the time between changes of state of the pin, we see it changing state every 6%ms, exactly as predicted.



&

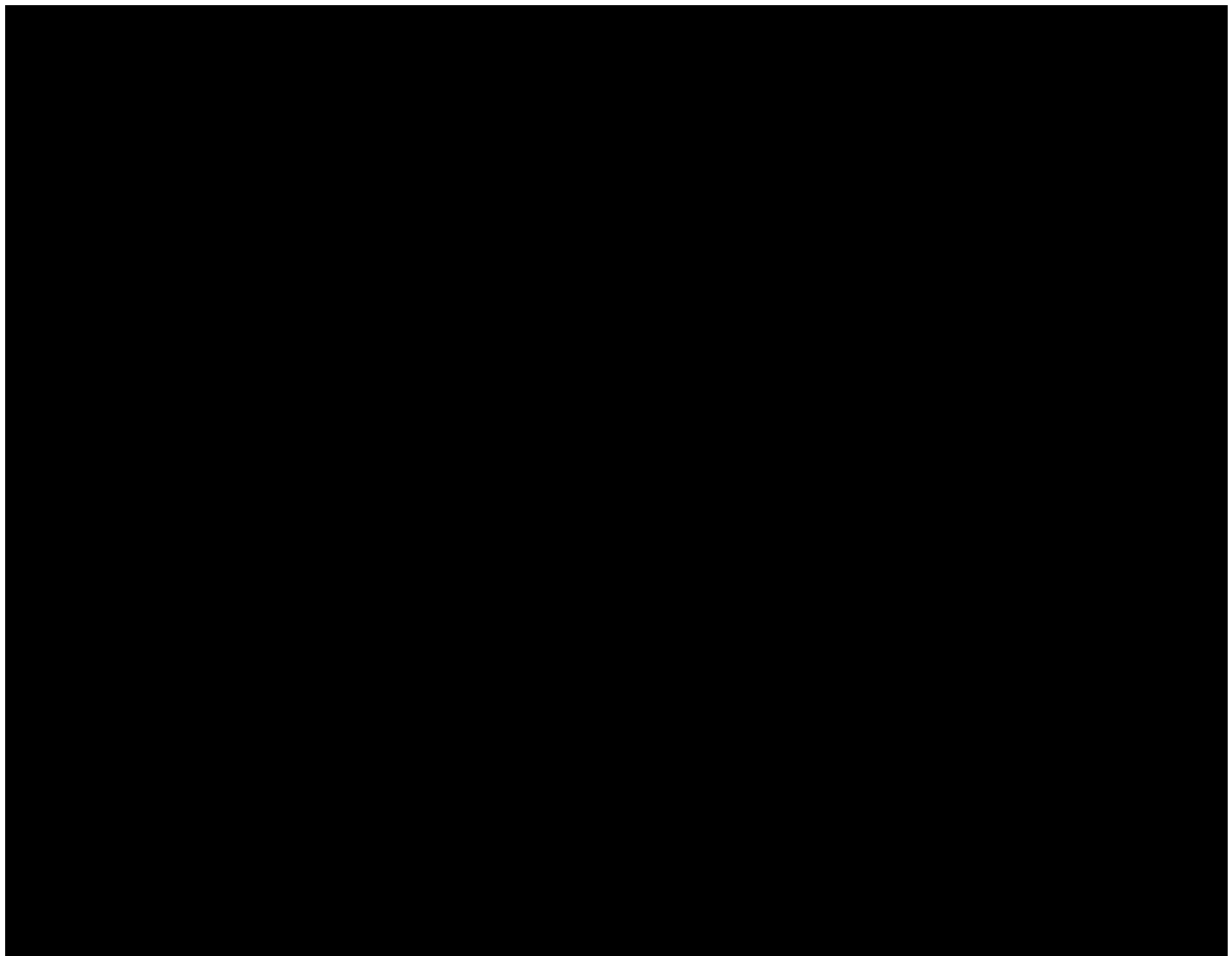
!ow about flashing a , -D at a rate of one flash per second, and at the

+

(he key5echo macro is exactly the same as the previous version of the program and just copies the key state to the corresponding ,D on the Atari# !)'00 board.

(he main program is the driving force, calling the ,D5(ask and key5echo task repeatedly in a forever loop. Also, it sets the )&9( \*, +it % pin before calling ,D5(ask and resets it when ,D5(ask returns. We can use this to measure the time taken for the microcontroller to run the ,D5(ask using an oscilloscope.

(he ,D5(ask sets or clears the ,D connected to )&9( +, )in F, using the ,D5(imer variable to count 6% # 6%ms, for a 6 second delay. (he ,D5(imer variable is decremented every 6%ms by (imer5%51>9 until it reaches zero. (he ,D5(ask changes the state of the ,D when ,D5(imer reaches zero, reloads ,D5(imer for another 6% # 6%ms and stays in that state until ,D5(imer reaches zero again.

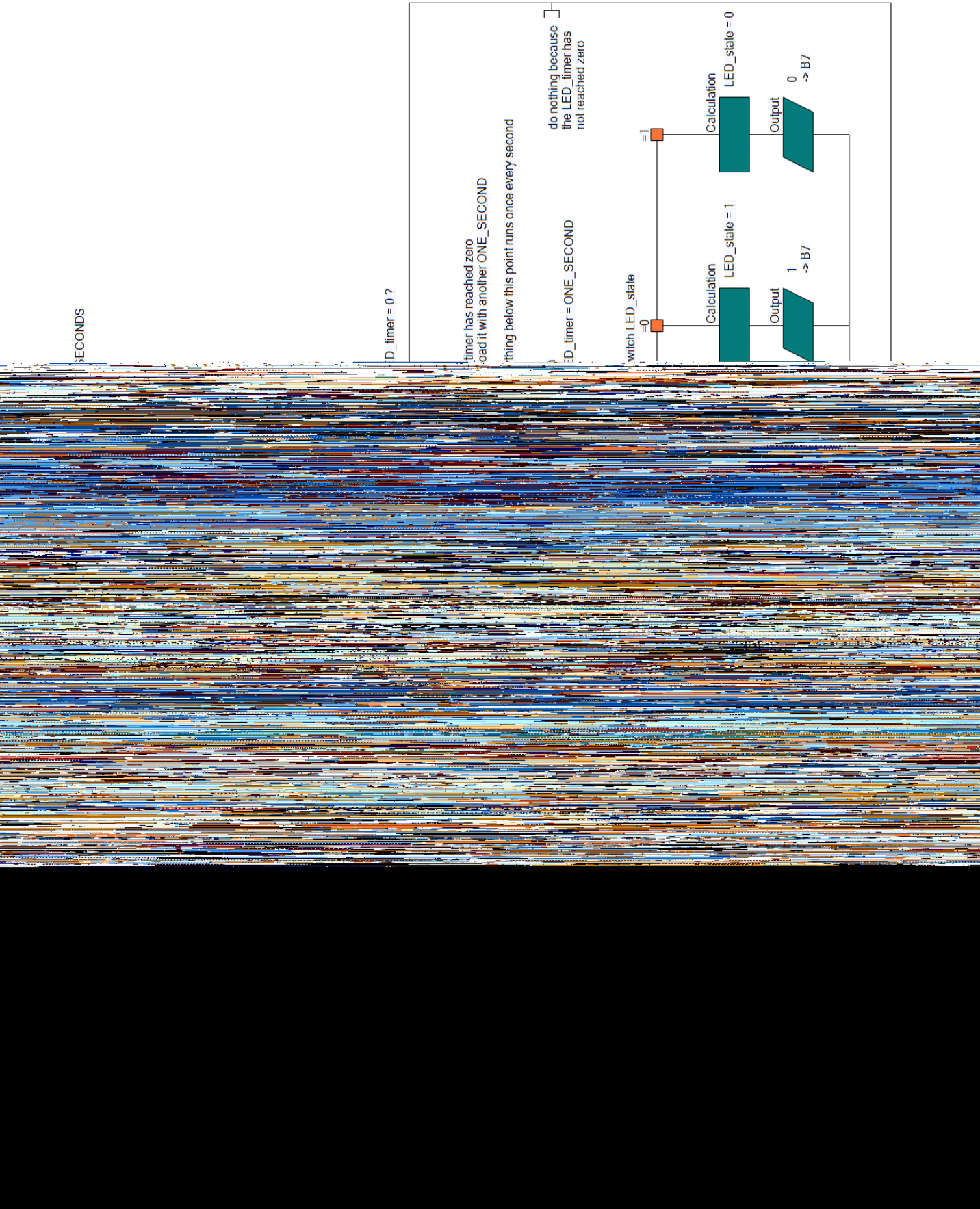


- &

- (1) \*reate a global variable 7e.g. , -D5(imer8.
- (2) In the (imer5%5l>9 macro4
  - a) "here the program reloads the counter 7every 6%ms8, decrement , -D5(imer until it reaches Gero.
  - b) If , -D5(imer is equal to Gero, do not decrement it.
- (3) In the task that uses , -D5(imer 7i.e. , -D5(ask8, set , -D5(imer to the timeout required in number of 6%ms increments 7e.g. 6%% # 6%ms I 6 second8.
- (4) In , -D5(ask, periodically test the value of , -D5(imer.
  - a) "hen , -D5(imer reaches Gero, your timeout has occurred and , -D5(ask can perform your ne#t task step.
- (5) If you were timing a "one off event", the interaction of the (imer5%5l>9 macro, , -D5(ask macro and , -D5(imer has completed the desired operation.
- (6) If you need to perform a "cyclic timing event", reload the , -D5(imer variable with the desired timeout value inside the , -D5(ask macro.
  - a) (he (imer5%5l>9 macro will decrement the , -D5(imer variable every 6%ms, until it reaches Gero.
  - b) In the , -D5(ask macro, periodically test the value of , -D5(imer. "hen , -D5(imer reaches Gero, your timeout has occurred etc etc.



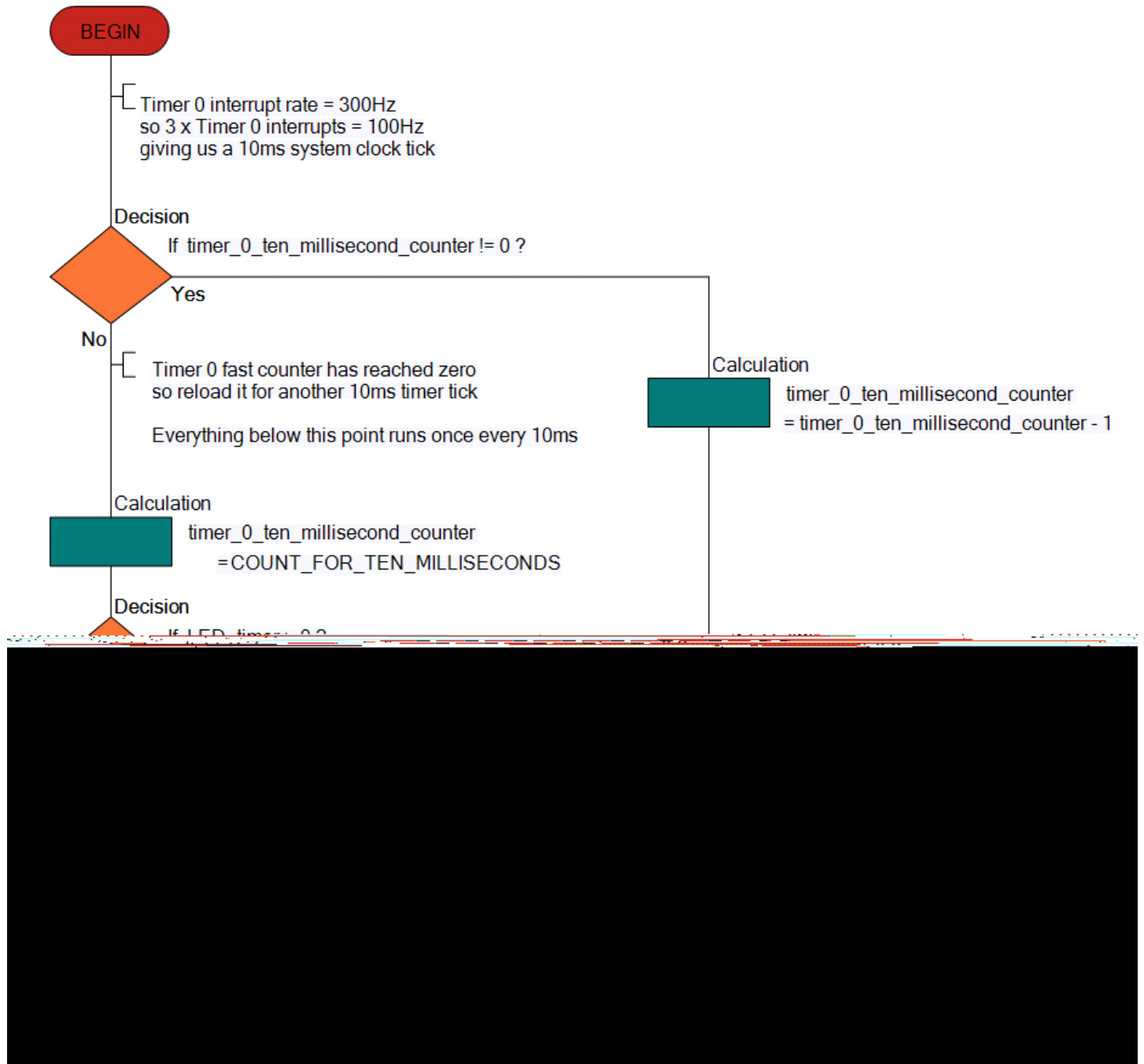
key5echo, main and ,-D5(ask interaction4



one5second5,-D5and5key5echo5) roq5A.fcf#

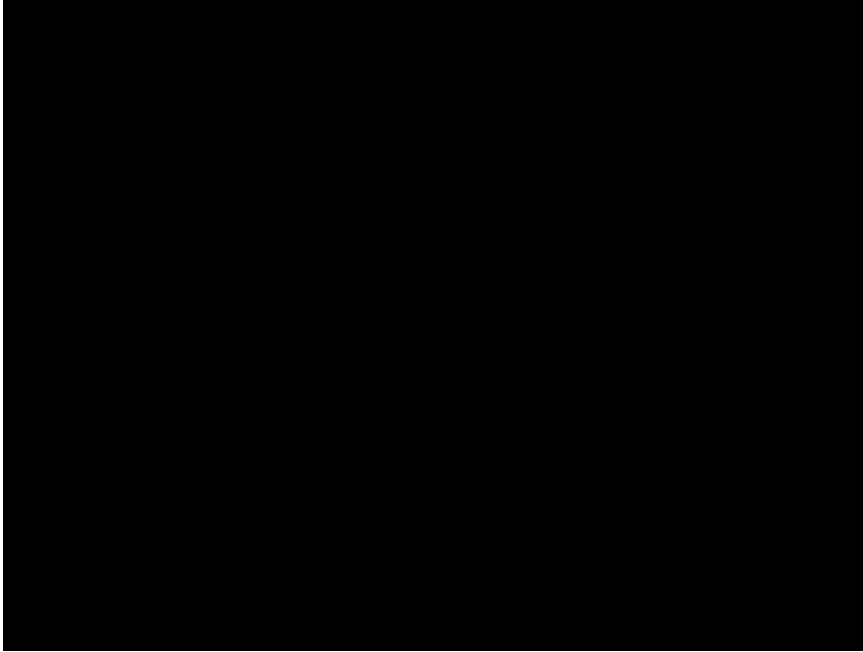
(imer5%5l>9 1 the real work happens near the end at "if ,-D5(imer 5 %", where it either decrements ,-D5(imer if it is greater than Gero or stops decrementing it when it reaches Gero.

### Timer 0 Interrupt Service Routine

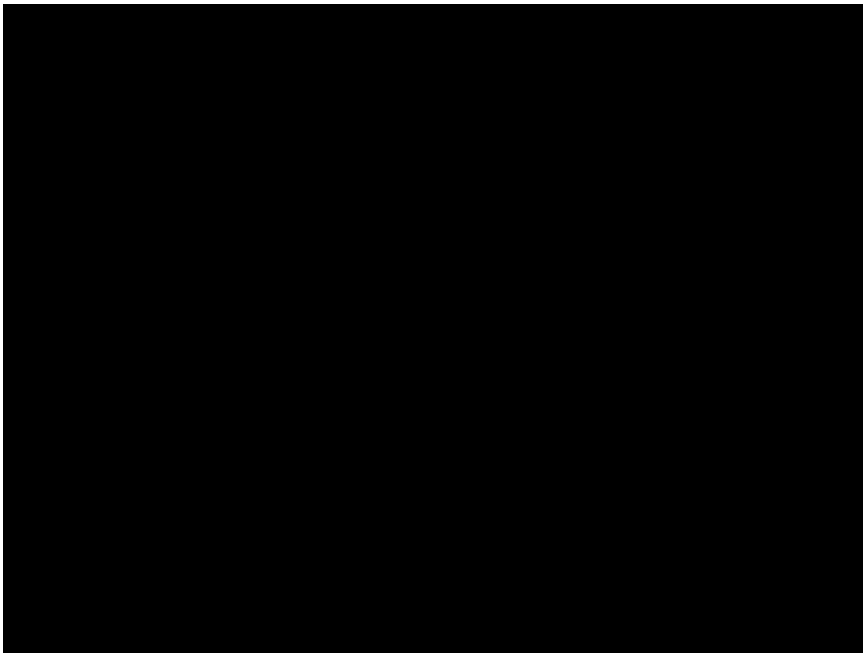


```
one5second5,-D5and5key5echo5)rog5A.fcf#
```

.  
\*  
)ort \*, )in % shows that the microcontroller only spends  $4\mu s$  in the  
, -D5(ask.



(he microcontroller spends  $16.2\mu s$  running the rest of the program  
7key5echo8 which is almost the same as when using "Delay"7  $17.2\mu s$  8.



.part from comple#ity, the timer interrupt timing method really hammers the Elowcode F simulator. .lthough I don2t know the inner workings of the simulator, I suspect that it uses "Delay" for Elowcode F "thinking time" and updating the models for the connected hardware.

' &  
.lthough more complicated to implement than a simple "Delay", the timer interrupt method frees up the microcontroller to perform other tasks while timing tasks runs in the background.

. mi#ture of interrupt timer driven timing measurements and delays for critical timing with normal "wasting time" Delays for non-critical parts of the program gives the fle#ibility for the design of really powerful systems using the atri# !)'00 board and Elowcode F.