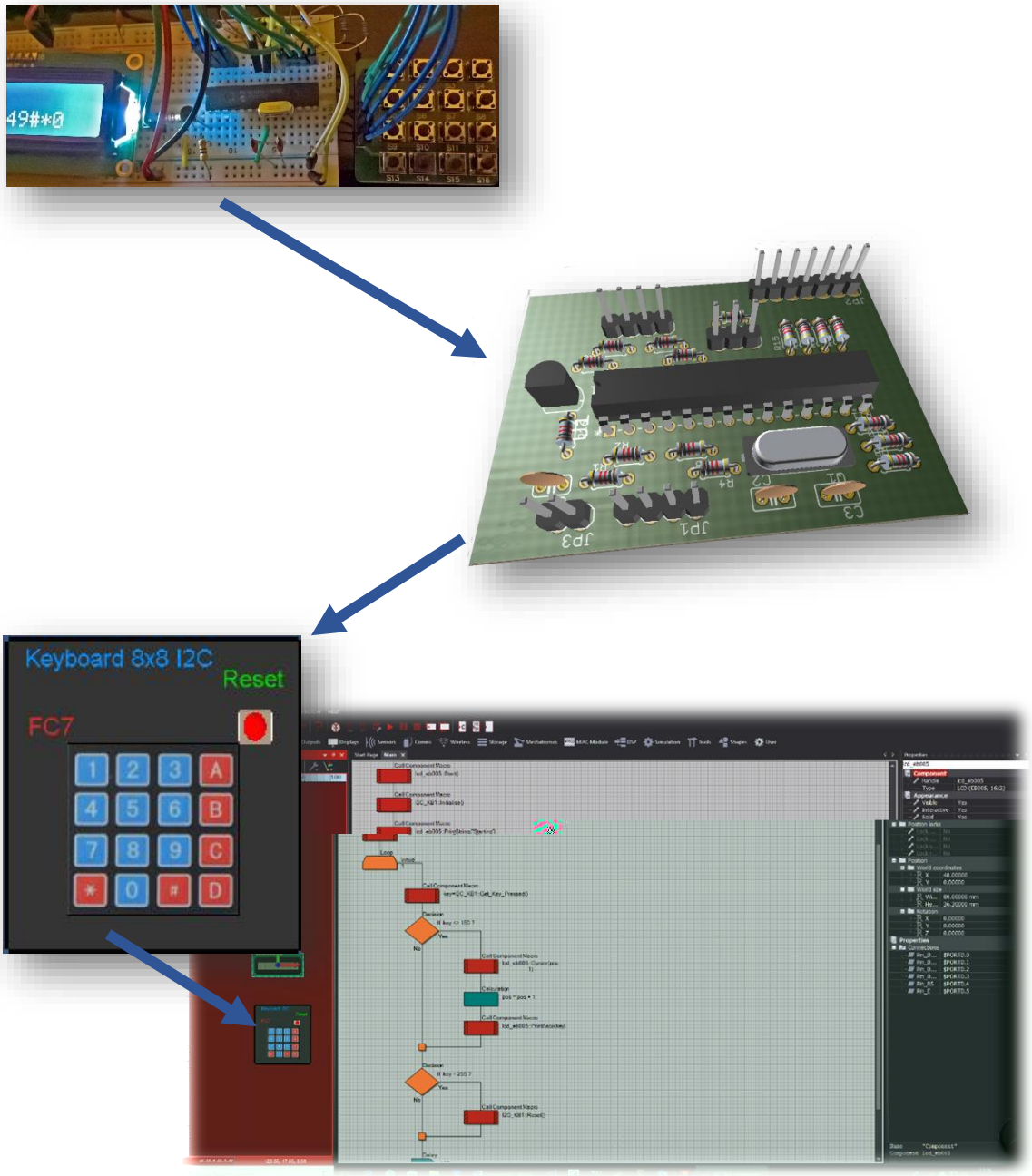
I2C Keyboard Controller

How to build an Intelligent I2C keyboard controller for Flowcode 7 Projects, extendable to 80+ keys.



Build an Intelligent Device &

The FC7 component to Control it

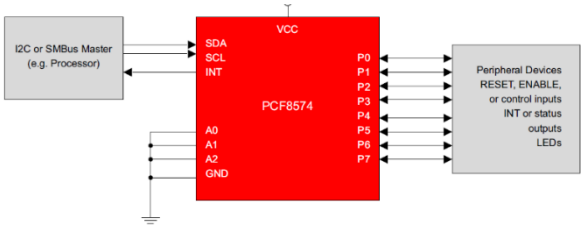
Developing embedded systems can result in some devices taking up a lot of resources such as I/O ports and/or program time to control them. One such example, that is often encountered, are keyboards. Lots of digital connections and program time to scan and decide which keys have been pressed. ***See Appendix B***

I want to show you, how you can take a micro-controller and program it to fulfil a specific repetitive function, that will enable it to be used as a specialist device for a FC7 project - a keyboard the device has the functionality, the component does not. ***See Appendix B***

This approach could also be adapted to fulfil any other repetitive task that you can think of. (invention is limited only by your own imagination)

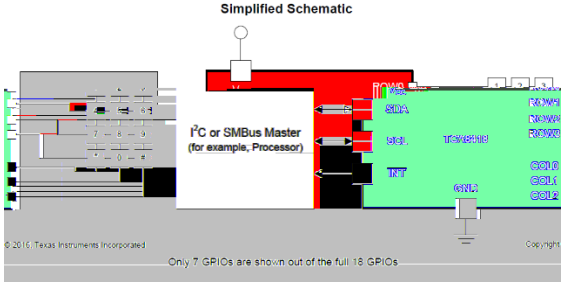
Using dedicated controllers, frees up valuable resources on your project (program memory and I/O pins) by doing the donkey work for you. Once you have built your specialist device, you can then build a Flowcode component to communicate with it. You are then able to concentrate on your project to do the more important stuff that you require. Flowcode is a powerful development platform, that will enable Mechatronic Engineers to build Intelligent devices capable of performing endless possibilities of specialist modular devices, that you simply plug into your main design project.

Consider this device PCF8574 I/O expander



This device looks ideal for a keyboard. It allows you to add extra connections that you can access via the I²C port. The problem is, it has no intelligent functionality at all. Your project MUST still do all the donkey work.

What about this one TCA8418



It has the best of both worlds. Fewer connections and some functionality. The problem now is, it may not have the required number of and/or functions or keys.

So, what do you do?

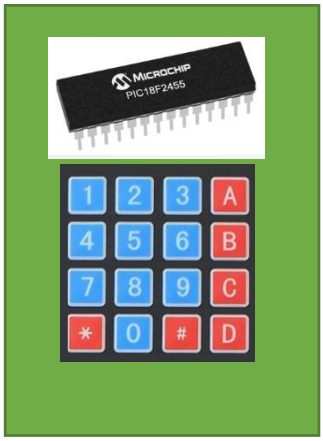
This tutorial will demonstrate how to build:

- An Intelligent I2C Keyboard Controller (expandable to full 80 plus keys, you even get to write its specification)
- A FC7 Component to control this new Device

This illustration shows you what is involved from a Mechatronic Engineers perspective. You will Build: -

The Device

You make it! (on a PCB)



Using a - PIC18F2455 (or other MCU)

You have the I2C Communication/ Connection you want

FC 7 Component

You Build - the FC7 Component to control it.

FC7 I2C-KB component

Of all the + they are designed to perform various functions such as; input/output expanders; LCD display controllers; flash memory controllers; ethernet bridges etc. What they have in common, they all probably started life as a standard microcontroller produced by companies such as Microchip.

Engineers, design devices using these standards microcontroller parts, to build intelligent digital devices, that you connect to make complex embedded systems, and then give them a unique name.

This tutorial will help you to design and build an intelligent device, that you have programmed yourself to fulfil a specific function, using standard Microchip parts.

Now you get to build your very own Intelligent controller - a keyboard controller. program it, then connect it to your project using your very own Flowcode component. This technique is not new. Modern computers are full of smaller specialist microcontrollers, programmed to fulfil repetitive functions, that frees the main CPU to get on with the more important stuff. Furthermore, you can repeat this for any other repetitive tasks.

STAGE 1 – The Device (I2C Keyboard Controller)

STEP 1

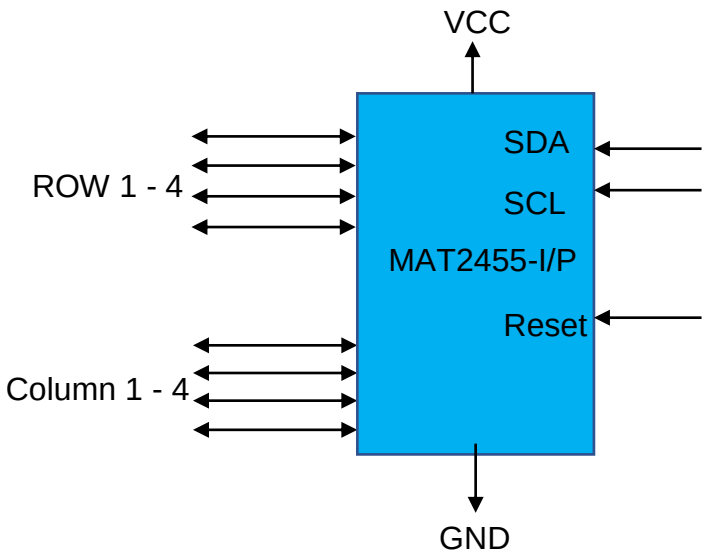
+

an industrial PID controller or CNC machine controller. They need some form of human interface such as a keyboard to input data. WE know they take up a lot of I/O pins and program memory to work efficiently. Therefore, it would be better to give this repetitive task to a dedicated keyboard controller. One that you have designed and built yourself. Now you will know exactly what is going on inside this controller and better understand how to communicate with it.

The routine to scan the keyboard will remain pretty much the same. Only changing when more keys are to be scanned or scan speed to be increased; key buffers added etc., increasing the options available to you. This only leaves the choice of Microchip or other controller to be used, hence your devices specification. Your choice!

Here is the device you are going to build a starter 4x4 16 key keypad controller, a basic starting point to begin with. It can be developed to handle up to 40+ keys. Change the controller and scan routine and you can extend this further to 80 plus keys.

Introducing your: MAT2455-I/P 16-42 key Keyboard controller for I²C Bus



Well, a PIC18F2455 -I/P. You are going to program it to perform the functions of a keyboard controller. If you were a company, you would get to name it and sell it, provided you have a FC7 professional license. MAT2455-I/P. Once you have selected/modified the code to be loaded onto the chip to suit your needs, you can call it whatever you want.

+-the-)

accessible. The 2455 is from the Microchip , or you could find an alternative if you wish, if you have the relevant FC7 chip packs.

STEP 2 – The Keyboard Controllers Program

The keyboard controller program has two parts

1. The I²C Communication Function Protocol (thanks go to Ben & Leigh at MatrixTSL)
2. The Keyboard scan routine

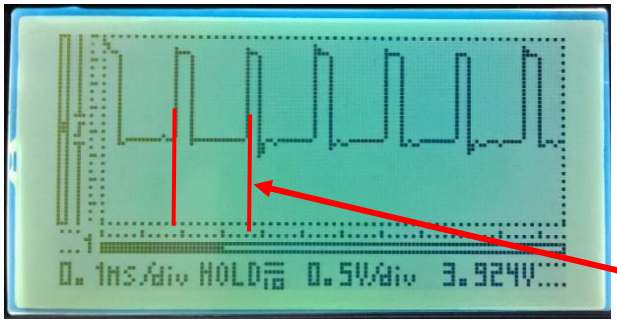
This basic routine has only four columns/rows that can be extended to include more if you wish 4x4/16 keys or up to 80 keys with a different MCU. Each column starts with one output being turned on whilst the others remain turned off, then checks to see if any of the inputs has gone high as the program steps down that column. Switch de-bounce is firmware controlled at the master I2C end. We know which key-switch is connected to each column-row configuration. Therefore, we can say exactly which key on the keyboard has been pressed within that column. That enables us to decide what that key does or represents) choose whist you have the routine loaded in FC7)
easily be transferred to your project using the I²C protocol. It is up to you how far you wish to develop this design.

The Scan routine is stored in a macro that is called from the main programs - loop) , the I²C service protocol **MUST** reside in the main loop, as any calls will push results onto the stack and must not become corrupted or over-written. In other words, it is at the beckon call of your projects **Master controllers I²C port (SCL) clock.** When the master calls, your keyboard controller must respond quickly, with a result of any key pressed, or a special return value to indicate its **STATUS (**150 - *ok but no key pressed***)**. Anything else (255) (*fault trigger a restart request).*

have a look at the keyboard program in detail. It takes under 1ms to complete a full scan including checking to see if the master I2C controller has called. That meets current modern keyboard poll times - <5ms. We need to perform tests such as diagnostic testing to ensure that the data is correctly stored in the variables, then use code profiling to check for efficient code.

Here is the result from a pocket - field testing scope. I had connected a probe to one of the column outputs and captured the pulse generated. Every pulse indicates the loop has completed a full cycle scan the keys and test for master I2C call (approx. 0.2ms/complete cycle for 16 keys scanned).

Note! All four columns are scanned before returning to the main loop.



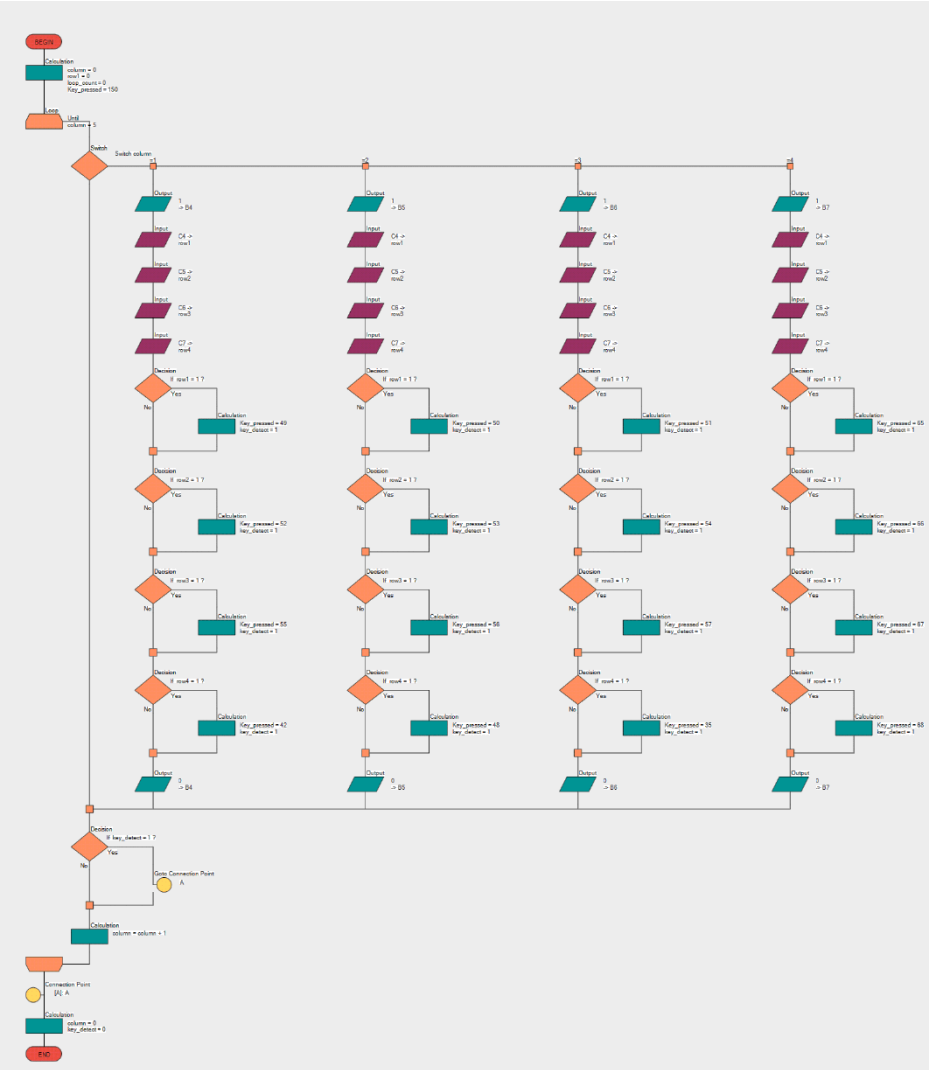
The scope also indicates how the) -

turned off correctly giving false positives at the row inputs. It should be a nice square pulse.

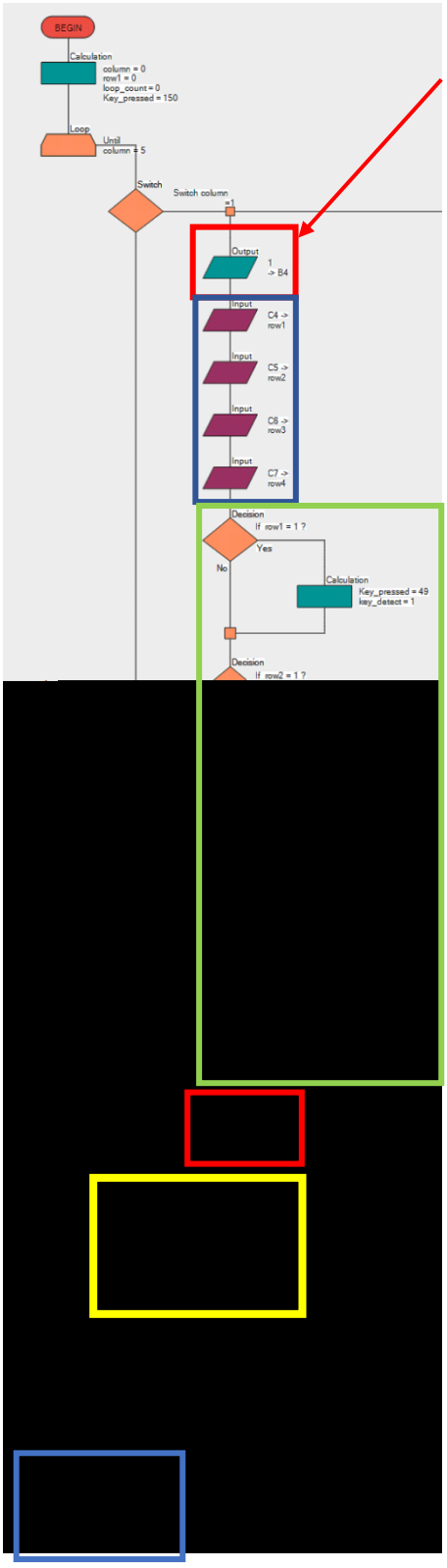
0.2ms complete cycle

Take time to study the scan routine using FC7. The scan routine is called from the main loop, it checks to see if a master call has been made first. Then the scan routine is started. From the top, the following variables are set.

- Column = 0 (increments by 1 every time a column has been done)
- Key_pressed = 150 (sets the return code to enable the master to check if no key pressed but the slave is still running OK. *If master calls, it gets told 150 no key pressed, otherwise the ascii code is sent*)
- Key_detect = 0 (set when a ***key-press*** has been detected, stopping the scan and jumping out of the loop. Improves speed and efficiency).



have a look at the programs column structure.



First, we turn **ON** an output. Do not use a **led** for an output on the Dashboard Panel. The logic operations do not like it errors will result! Why, to do with the hardware on the chip. It requires **weak pull-up** ed on for the I2C to work on the **B port – LED's conflicting with the I2C port!**

Second - check each input for logic level and assign to a variable **row1; row2 etc**

LOGIC - We know, that whilst we are in this column, any input found (the variable will store as a **1 or 0),** to be high or low, due to the switch matrix that will correspond to a specific switch being pressed.

The switches in column1 are **1; 4; 7; ***

ASCII – 49; 52; 55 and 42

The decision checks the logic value for each rows variable, if true, then assign the appropriate ascii value for the **key_pressed** variable (this over-writes the 150), ready to be transferred by the I2C port. The variable is stored and carried out of the macro back into the main routine safely.

Turn off the column output.

Now some time saving. If a key is pressed, why continue to the end of the scan? It will be much more efficient to stop scanning now, we have found that a key that is pressed and therefore we can return to the main routine.

Finally, reset the **column** and **key_detect** variables before exiting the macro. This will enable the next scan to begin as normal.

All the other columns will be the same functionally, but different columns outputs turned on and ascii values assigned for the keys pressed.

STEP 3– I2C Protocol – *Thanks go to Ben & Leigh at MatrixTSL*

The 2 **Comms'** pack I2C slave. Building a device using this method of communication can be difficult due to the high frequency, **EMI** Electromagnetic Interference on the HF radio frequency band (100kHz). Wires must be as short as possible or kept well apart from each other. Capacitance is our enemy.

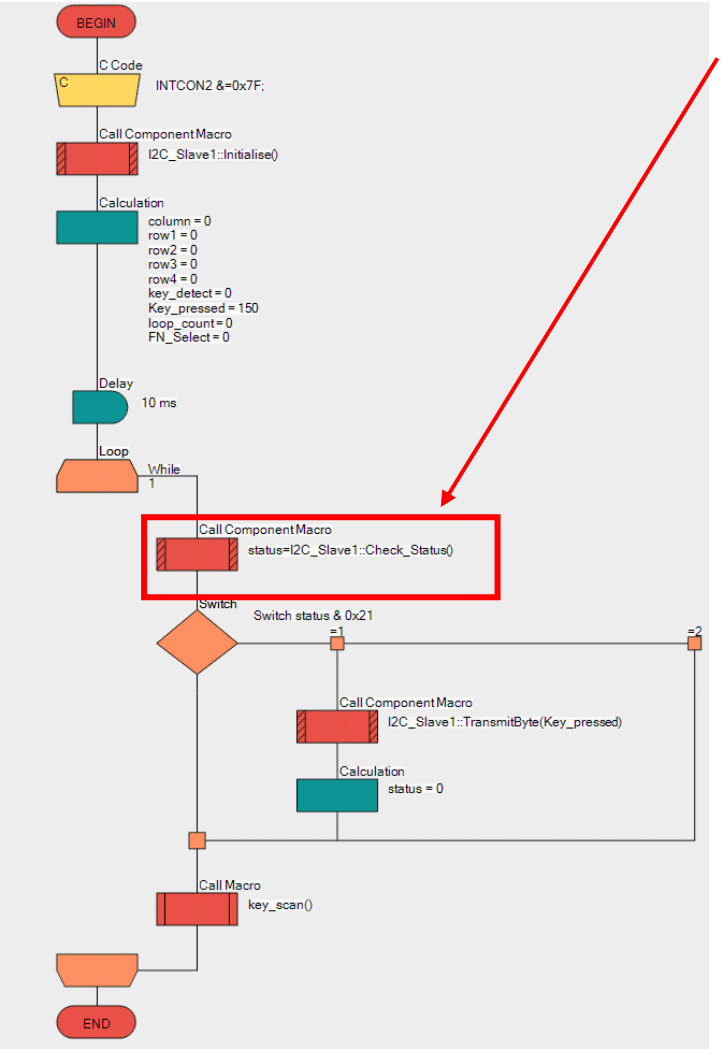
DO NOT twist cables together from the output.

Short single solid core wire is ok but for optimum performance use mini RF cable (mini coaxial). This is where the outer screen has an air gap insulator to kill the capacitance between the solid core and the screen. It is possible to transfer data over 1k metre using a driver such as P82B96 Dual bidirectional bus buffer and eliminating many of the above problems.

C

internal weak pull-up resistors. - tors of the value range approx. 2k to 10k

The main loop comprises the following:



Status this checks the I2C hardware. If a master I2C call has been made the internal status register will be set as follows: -

Bit 0 = 1 Indicates address/data byte available in the buffer to read

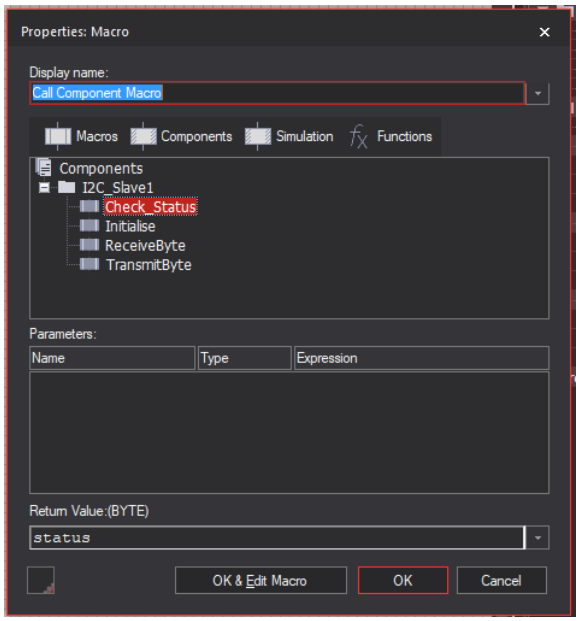
Bit 5 = 1 Indicates that the last byte received or transmitted was data (else address)

- 0x21 hex value (Bit 5 & 1 are set)

If this **status macro** returns with this value 0x21, the **Switch** directs the program flow to =1 position

In other words, the master is waiting for data from the slave. The slave does not and cannot drive the **SCL** clock. Only the master oversees the clock. Therefore, the master drives the clock line and the slave outputs the data in sync with the clock. You are NOT allowed to write to or read from the register whilst this is happening. Any attempt to place data in this output buffer during a data transfer will result in stalling the I2C hardware. All the clever stuff is going on in these two

macros produced by MatrixTSL. The switch=2 is for catch all other possibilities prevents errors.



The status macro there is a lot going on within this macro. It reads the internal I2C port status register called **SSPSTAT**

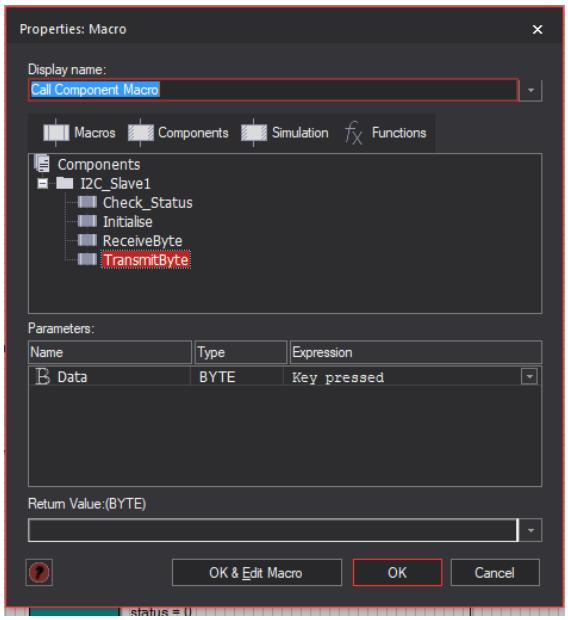
bit 0 **BF:** Buffer Full Status bit
In Transmit mode:
1 = SSPBUF is full
0 = SSPBUF is empty
In Receive mode:
1 = SSPBUF is full (does not include the ACK and Stop bits)
0 = SSPBUF is empty (does not include the ACK and Stop bits)

bit 5 **D/A:** Data/Address bit
In Slave mode:
1 = Indicates that the last byte received or transmitted was data
0 = Indicates that the last byte received or transmitted was address

Returns the value in the variable - **status**

The keyboard I2C device must respond quickly to deal with the master request poll time. Most modern keyboard have poll time 5ms or less.

The data from the key_pressed variable is handled by the macro TransmitByte

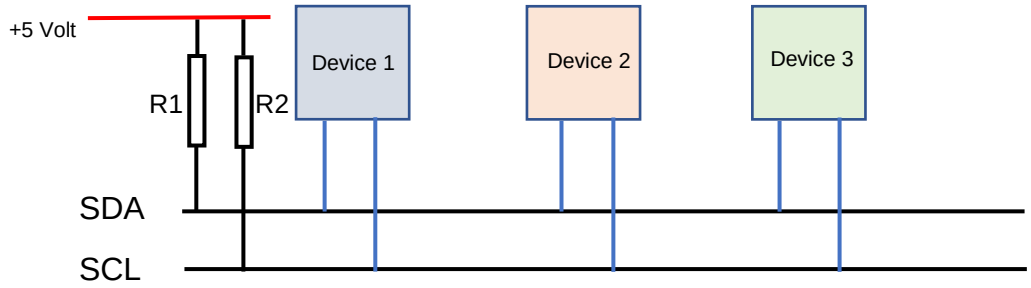


Once the status has been processed and found to have valid address data, the status BITs are set, allowing the switch to route the program flow. This next macro begins transmitting the data to the master controller. The component we have designed is expecting only one byte of data, therefore, our keyboard controller conforms to this protocol.

Exiting this macro, we must reset the status variable to zero to ensure we read the next request from the master correctly.

The I2C Bus

The I2C bus consists of two wires connecting each device together on the bus. These lines must be held high, using resistors of the correct value. To calculate the resistor values, it is advised to follow the correct I2C standards.



The temptation to put pull-up resistors on each device MUST be avoided, as this would effectively create a parallel resistor value, much smaller than may be allowed.

The maximum load an I2C device can switch at its output is 3ma for 5-volt devices. Therefore, we must ensure that the resistor provides the correct current vs capacitance (below 3ma load resistance).

Note! Fast and Standard modes are 400kHz & 100kHz

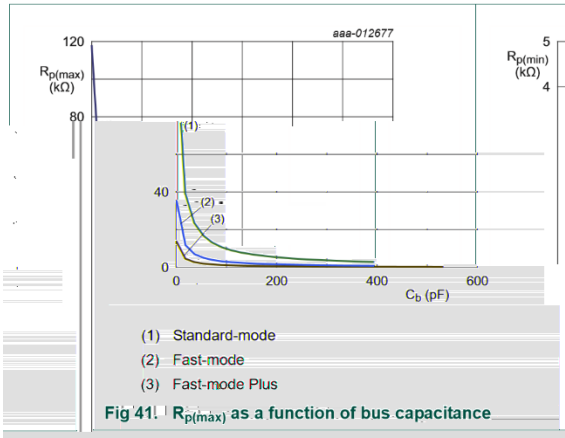


Fig 41. $R_{p(max)}$ as a function of bus capacitance

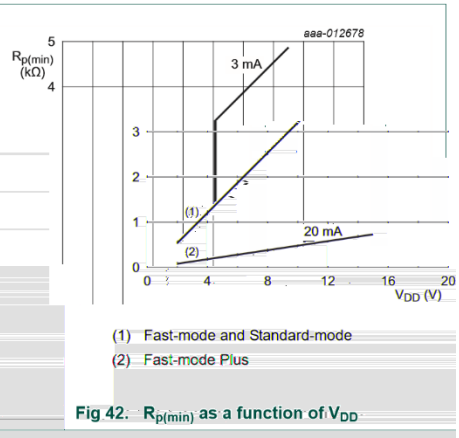


Fig 42. $R_{p(min)}$ as a function of V_{DD}

Think of it this way. A capacitor is like a storage tank. The bigger the tank, the longer it takes to fill it. If the wires you are using have a lot of capacitance measured in pF (picofarads), then the fast switching of the outputs takes time for the voltage to rise and fall. The graph on the left shows (1 standard mode 100 kHz) small capacitance allows higher resistor values. The graph on the right (1) shows VDD voltage means smaller pull-up resistor values to achieve 3ma. From this we can use this calculation as a good starting point.

Resistor value = VDD minimum 0.4 volts ÷ 3ma

There will always be some capacitance in your design, so you can assume VDD minimum to be less than the VDD power rail. This is because, when switching begins, the pull-up resistor never quite reaches their full voltage until the switching has stopped. Therefore, for a 5-volt supply rail, we assume VDD minimum to be 4.5 Volts. This gives us the following numbers:

Minimum Resistor value = 4.5V 0.4v

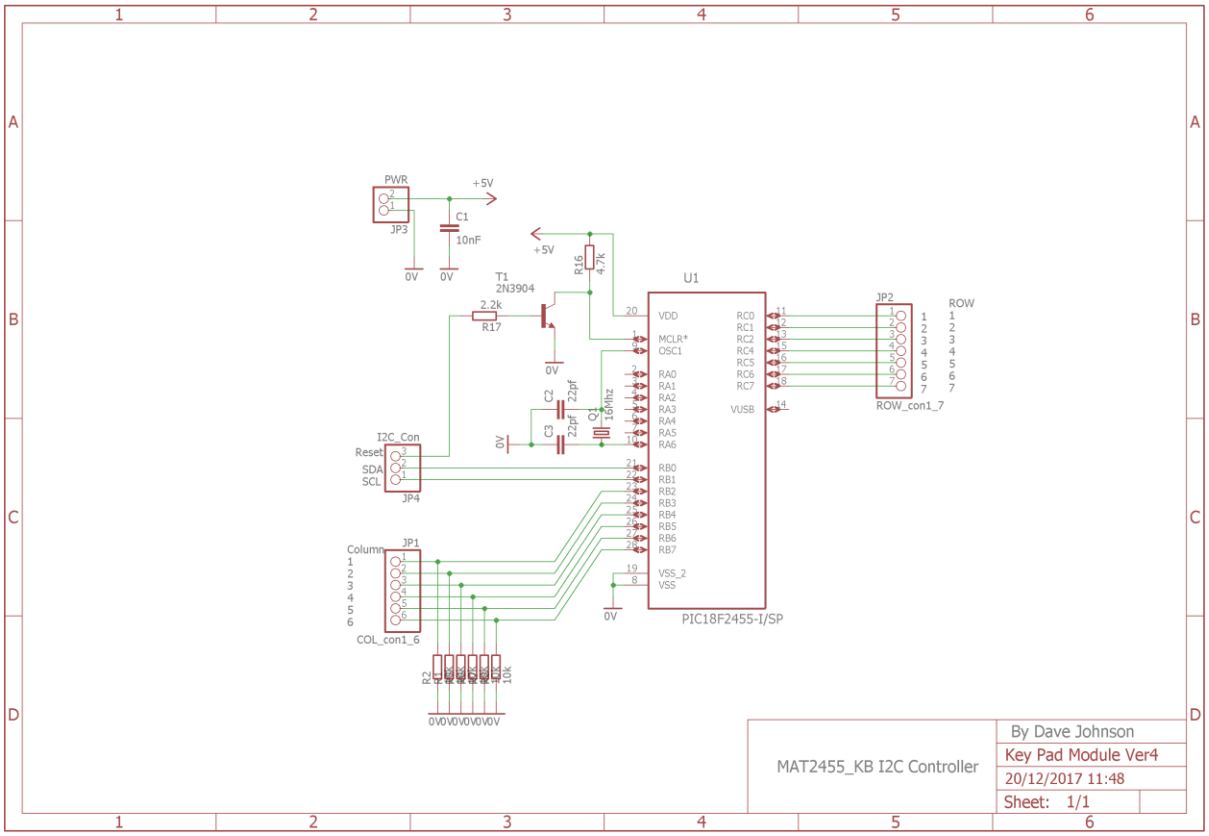
0.003 (maximum current allowed)

= 1366

,)0 - /

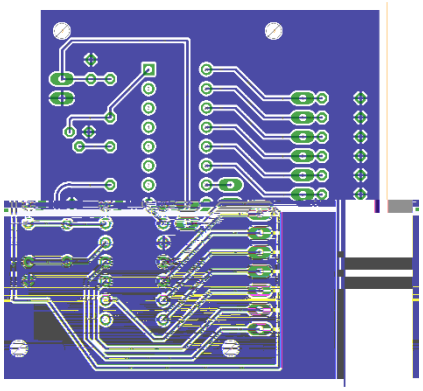
+ - absolute minimum to begin testing. You can increase this if you have low capacitance in your design.

STEP 4– Keyboard Circuit Design



The circuit is designed around a PIC18F2455 I/P controller running at 16MHz parts list is included in the appendix. I have allowed the design to accommodate up to 6 columns and 7 rows giving - 42 key keyboards. The pull-down resistors 10k must be included to ensure reliable operation.

The transistor is used to reset the keyboard controller when connected to your projects via FC7 keyboard component. It can be called from within the FC7 component - I2C_KB.fcpx.

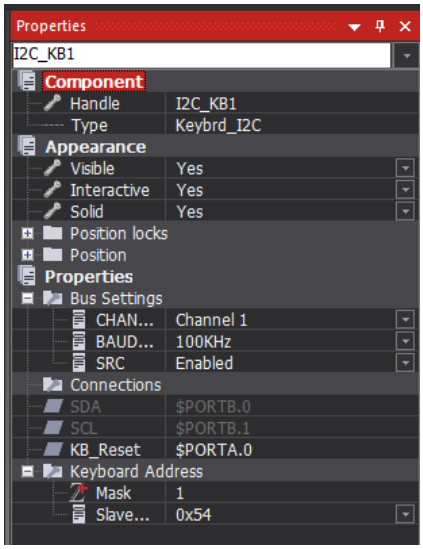


The PCB is a little bigger than necessary to allow for through-the-hole components to be accommodated. If SMD were used, it would be much smaller. I have included all the necessary files, as a download.

STEP 4– Keyboard Component – I2C_KB.fcpx

Once you have completed building the keyboard controller hardware, begin building the FC7 component. The properties available are:

Channel	hardware or software
Baud rate	100kHz; 400kHz &1MHz
SCR	Slew rate control
KB_Reset	
Address range	0x54 & 0x55
Mask	True (1); False (0)



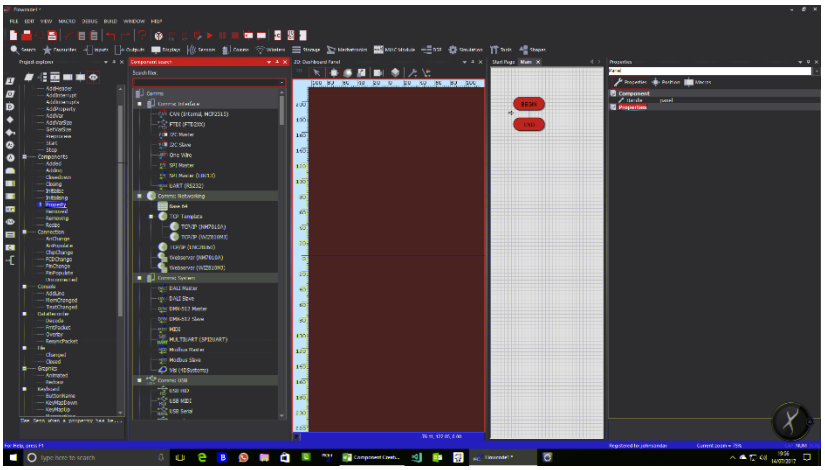
The component has been tested using the Microchip I2C hardware successfully at 100kHz. The address is available for you to change within the project for component creation, allowing changes to suit your requirements. The slew rate control feature, enables the I2C hardware to adjust the baud rate by monitoring the rise and fall time of the clock, remember that capacitance problem! It tries to overcome some of the problems it causes. The **mask** enables ONLY the correct address to access the device.

The reset pin allows your project to reset the keyboard controller, should a problem with the I2C communication occur (returned value = 255). Using the reset macro, will make the simulation LED flash. All this is handled from with the component creation.

STAGE 2 – Keyboard Component Creation

STEP 1

You begin by starting a new project. It is not necessary to select a particular controller, it is the embedded macros and their properties that are stored in the component file when finished. ***See Appendix B***



The screen looks something like this when you begin. From left to right:

Project Explorer; Component search; 2D Dashboard panel; System Panel; Properties Panel

The Project Explorer lists all the ***Events*** that will be associated with your finished component. A Flowcode 7 simulation event is similar to an Interrupt but aimed at performing a specific job. This is not just what you see on the screen but what may be downloaded onto the chip. Example, some devices require they are ***initialised***, i.e. sent command codes to get them going. This is where you set them up. Looking down the list, and there are many, select ***Initialise*** under the heading components. This is necessary for the component we are using a base **CAL I2C**. You will find this by using the component search feature in the next panel.

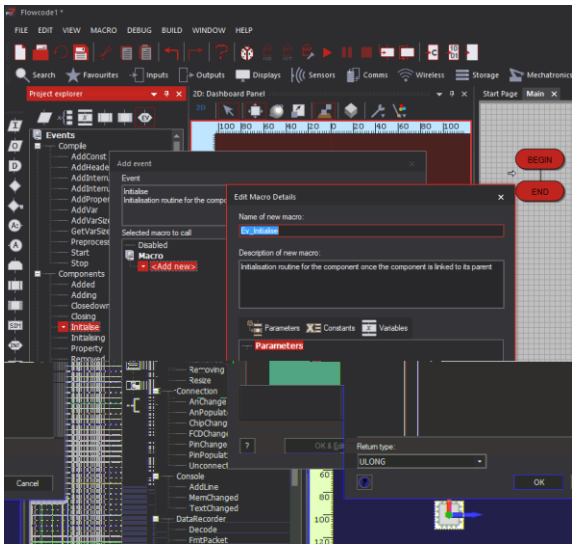
Component Search never sure where to find what you are looking for? The search tool makes finding what you want easy. Find the CAL I2C base component and place it onto the 2D Dashboard Window.

System Panel All the communication/functionality required for the keyboard component will be developed here, in order to control the keyboard device you have just built , from within your project.

Properties Panel The component is built by you using macros in the workspace window. You simply drag, whichever FC7 icons from the tool bar, onto the workspace to build the functionality you require for the component, just as you would for any other project.

STEP 2

You should now have the base CAL I2C component placed on the 2D Dashboard. Close the component search panel to give yourself more space to work with.



From the Project Explorer **<EV>** select ***Initialise***

From the dropdown menu select **edit** and then select **Add new**

You will be presented with the second box called Edit Macro details with the title called **EV- Initialise** with the Return type - **ULONG**

Click OK

You can now close the project explorer window.

This event will setup all of the links to make your component I2C communication work, once it is exported as a finish FC7 component.

STEP 3

Over on the other side the Properties Panel, are the keyboard component properties that need to be set up, with the connection pins named (these will be written to the EVENT Property). If you click anywhere within the 2D Dashboard Panel now, will show the current components properties are empty. What you are about to do, is give the Keyboard component the necessary links to ***Initialise*** the functionality you require the properties and digital pins, to connect your project to the keyboard controller.

It is very important to remember, without the components properties set up first, you will not be able to start using the FC7 flowchart icons and macros as nothing exists yet for them to control.

These are the properties we require:

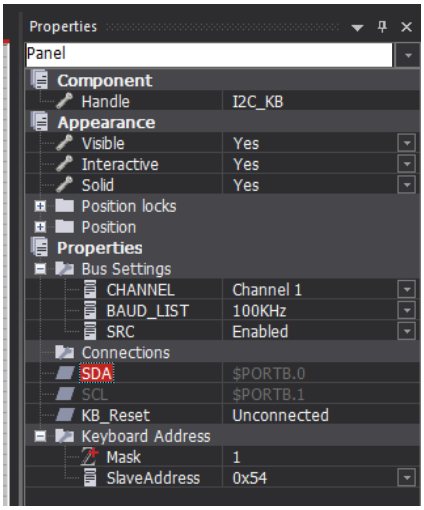
- CHANNEL
- BAUD_LIST
- SCR

These are the connections we require:

- SDA
- SCL
- KB_Reset

The I2C Keyboard Address:

- Mask
- SlaveAddress

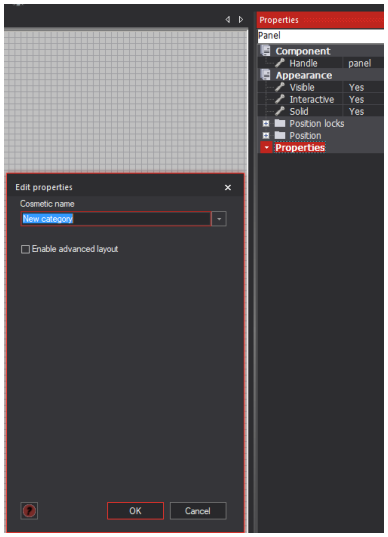


STEP 4 – Properties Panel

You will notice under the heading Properties are the headings for each sub category. Set them up as follows:

From the pull-down menu select New Category

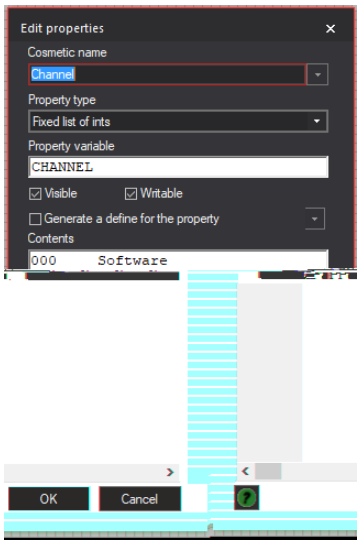
Give it a suitable name (Bus Settings), then ok.



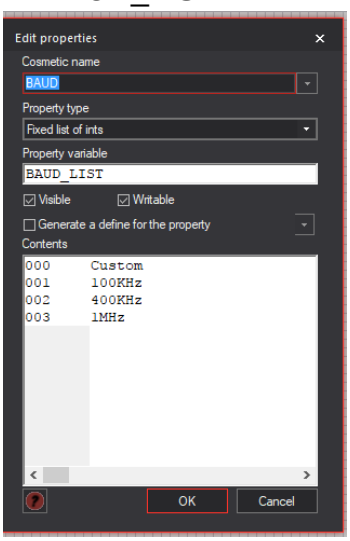
Now you can add each of the properties for this group.

Select add new and complete each box as show below.

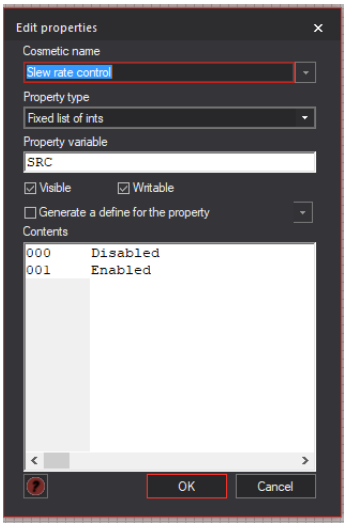
CHANNEL



BAUD_LIST

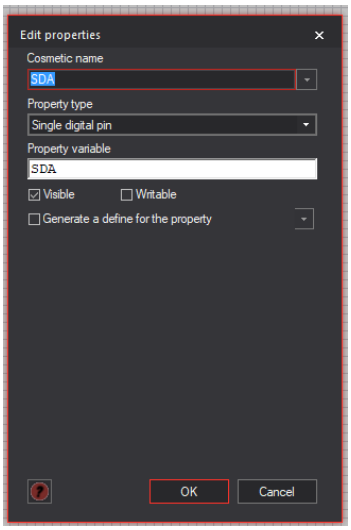


SCR

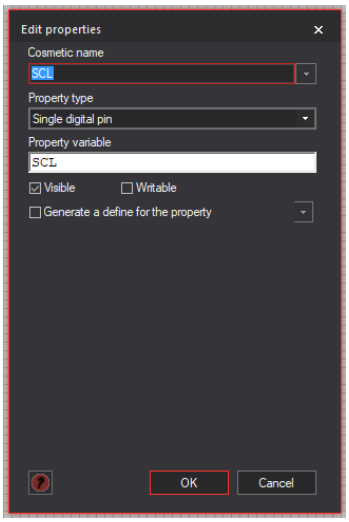


Next, New Category - **Connections**

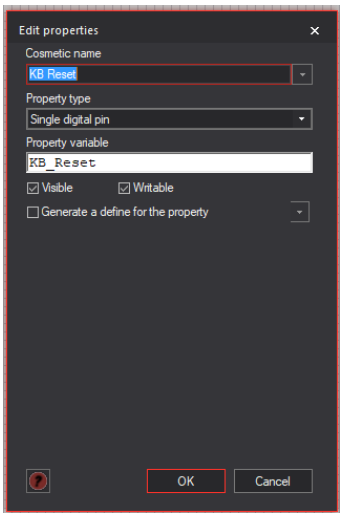
SDA



SCL

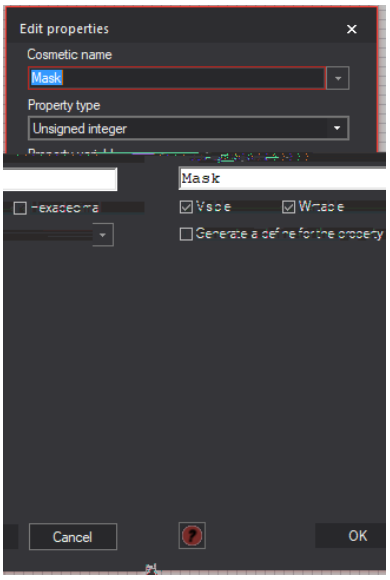


KB_Reset

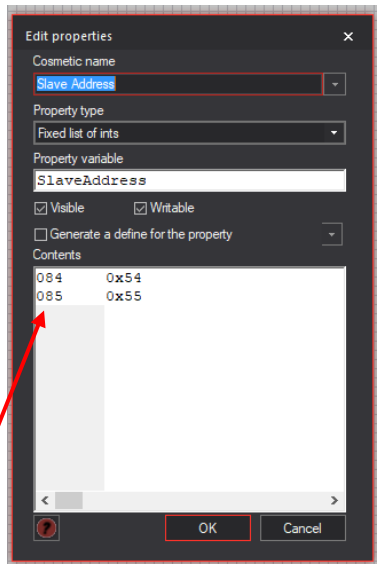


Next, New Category Keyboard Address

Mask



SlaveAddress



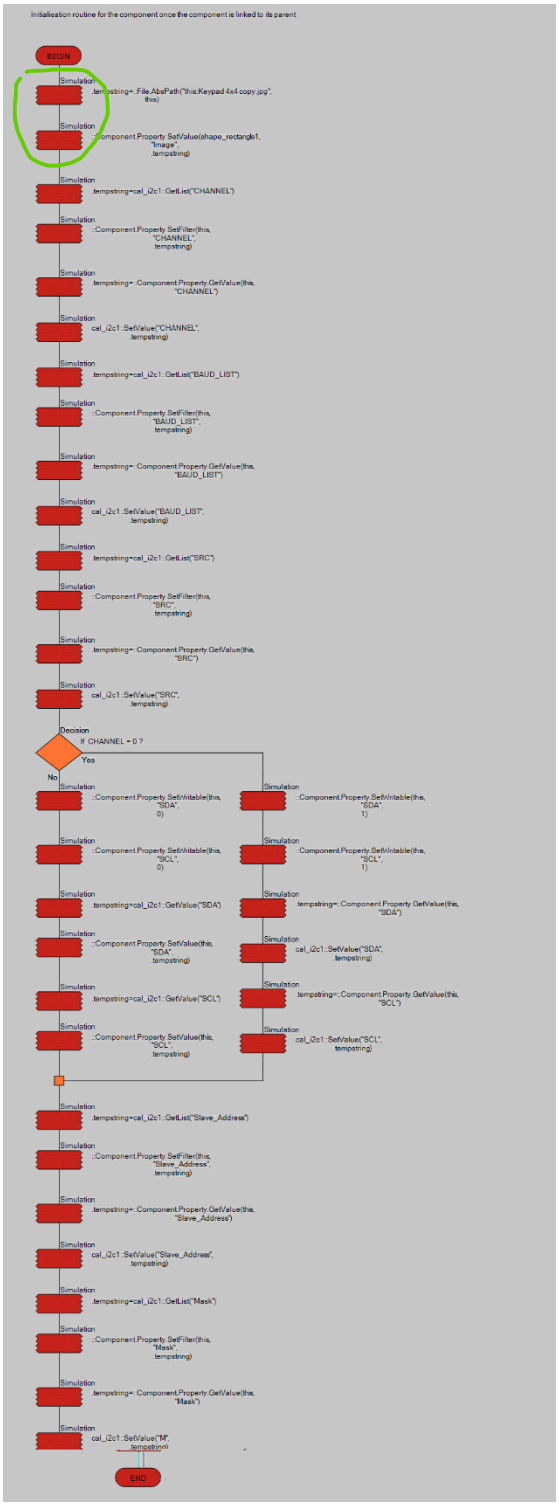
NOTE the address **contents** are the decimal values for the I2C address to be used by the Event **Initialise** be careful!

Now for those Macros

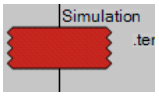
STEP 5 – Macro - (Ev_Initialise)

There are five macros for this component:

Ev_Initialise; Main; Reset; Ket_Key_Pressed; Initialise



The first two icons, ensure the keyboard image displays correctly. Future releases of this component may remove these two macros.

This macro 

is used to develop the functionality we require for our component. The Flowcode 7 API, uses it to link everything together, not just to simulate the on-screen graphics. We will be using it a lot. It is like writing the #C code we need, made simple.

The variable used is a local variable called - **.tempstring**. It is used to carry the linking data from the base CAL I2C to our component. We MUST use this variable in a very structured way. We must do one linking event at a time until that event is finished BEFORE starting another. Otherwise information/data will be lost.

Lets have a closer look at the above **EV_Initialsie** event

Example **CHANNEL** The information/data flow is as follows;

1 **Get_List** from CAL to .**tempstring**

2 **Set Filter** from **.tempstring**

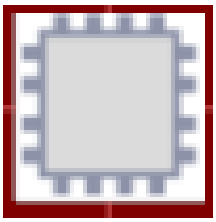
3 **GetValue** for CHANNEL from component to **.tempstring**

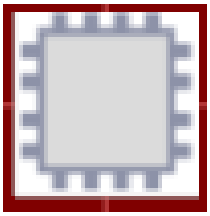
4 **SetValue –** for CHANNEL to CAL from **.tempstring**

2. Component

user makes changes
to achieve correct settings

3. Component





1. Base CAL I2C

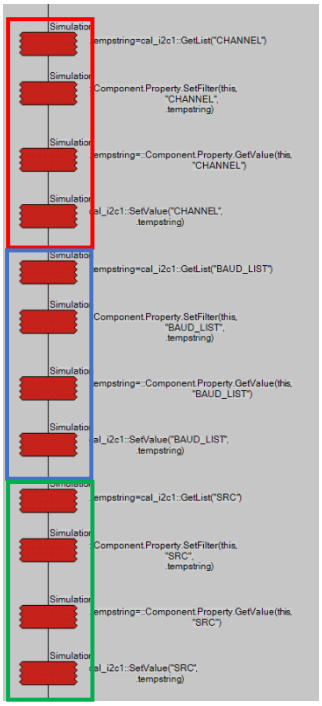
4. Base CAL I2C

This shows we need four simulation icons to complete the necessary data transfer. When the component is placed into a project, any changes the user makes are automatically transferred back by FC7 API, to the base component and the MCU set by the user. It is therefore Dynamic! The base component MUST NOT be in a different state to your component.

1	2	3	4
Get List from CAL	Filter out CHANNEL	GetValue from component Changes	Set changes to CAL

We do this, (repeat for **BAUD_LIST** & **SCR**)

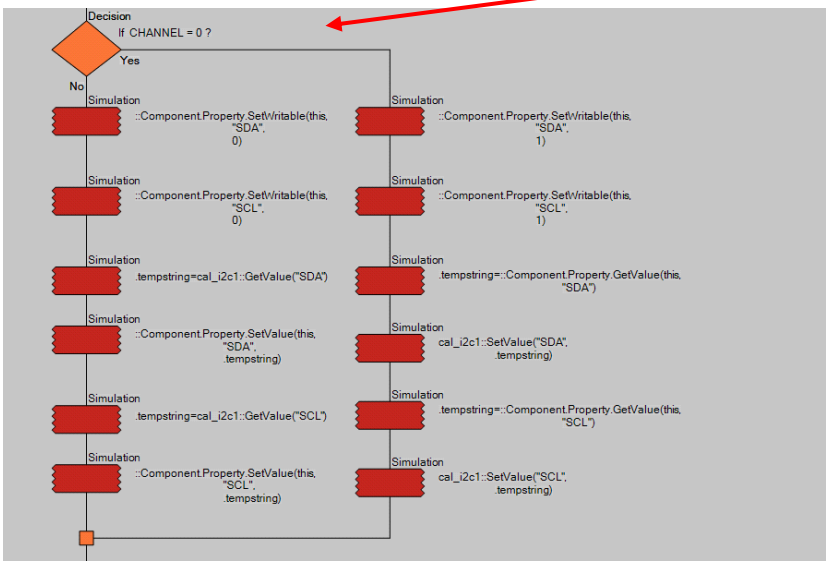
- **CHANNEL**
- **BAUD_LIST**
- **SCR**
- SDA
- SCL
- Slave_Address
- Mask



Repeat the above steps for all of the other component settings listed above.

Note - The KB_Reset pin is handled in a unique way as you will see later.

However, if the channel is set to software it is necessary to allow different pins on the project MCU to be selected. This is achieved using a **switch function** for:



SDA

SCL

Study the FC7 device project to familiarise yourself with devices functionality. This method is repeated for many of the communication macros used in flowcode!

This action applies to the **SDA** and the **SCL** pins as shown above. When the user selects different pins to be used, Flowcode API needs to know where to route the data back to the base CAL_I2C.

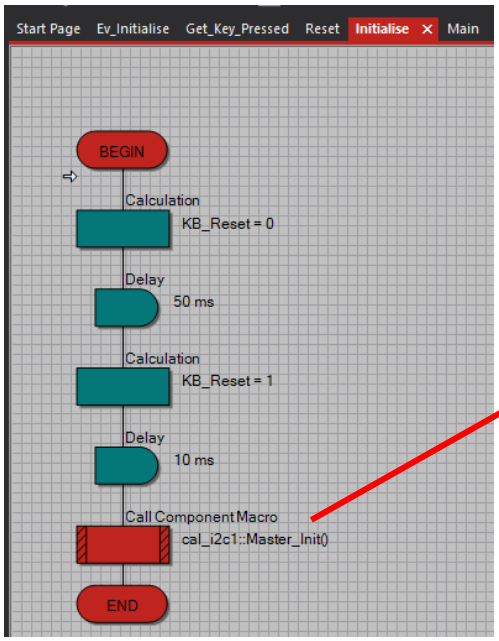
Now do the **four** linking simulation macros for the:

- **Slave_Address**
- **MASK**

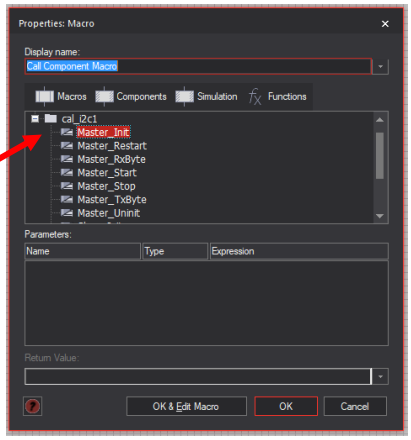
This leaves the properties to be set-up for:

STEP 5 – Macro (Initialise)

Now we begin to set up the component macros for the user to download into their projects. However, The **Ev_Initialsie** shown above will be hidden.



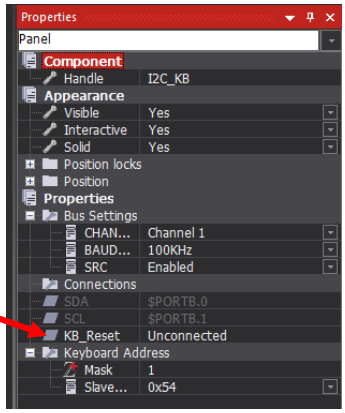
First Macro - **Intialsie,** we use this component macro to Initialise the base Master I2C CAL port.



Select ***Master_Int***

This will ensure that the project is correctly configured to set the base CAL as **MASTER** with all the correct setting you have included in the Ev_Intialise event are included.

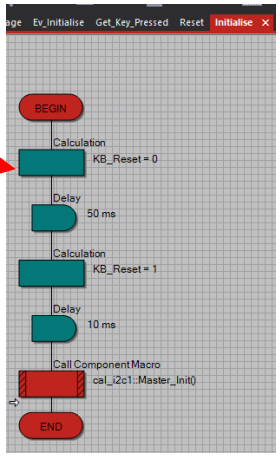
Remember that pin we have allocated for the reset KB_Reset



The variable is called KB_Reset

Whichever pin the user has selected for the connection on their MCU for KB_Reset, we activate that PIN as follows. The variable assigned to the pin is called KB_Reset.

- if we give that variable a value of = 0 the output is low
- If we set the variable a value of = 1 the output goes high

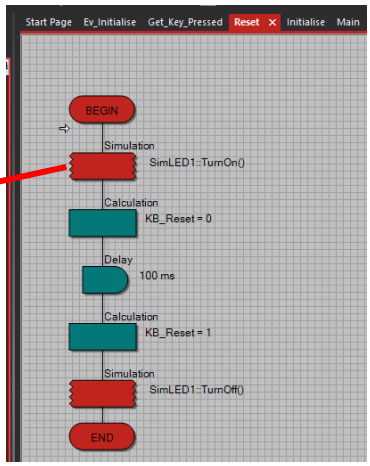


The icon used to achieve this is an ***equation icon*** as shown here.

Now we have controlled how the Keyboard device is setup during the initialisation period i.e. the master is up and running first before the slaves I2C port starts.

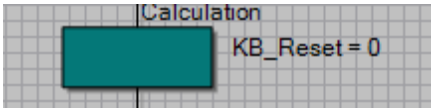
STEP 5 – Macro Reset – called by user if an error occurs

On the keyboard component a Simulation LED has been placed. It will flash when the Reset function is called by the Reset macro.

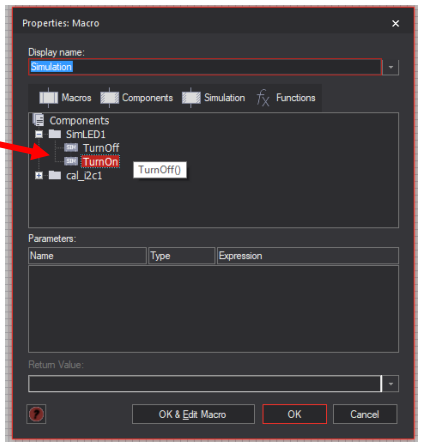


Use the component search to find the Simulation LED.

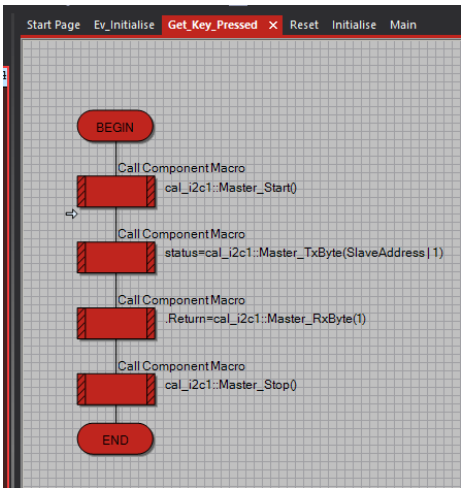
Use the equation icon to set the output for the KB_Reset PIN. OUTPUT = 0



When you place the simulation icon you will be able to set its value



STEP 5 – Macro (Get_Key_Pressed)



Use the base CAL I2C macros to obtain the correct functionality for transferring data from the keyboard)

It I2C protocol always begins with

- Start
- Master Transmit address of slave device
- Receive byte **.Return** variable
- Stop

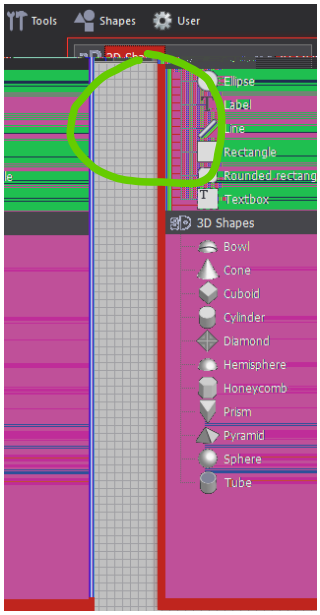
MatrixTSL have developed the I2C protocol macro to function correctly.

Not always easy to remember the correct sequence, therefore our component does it all for you!

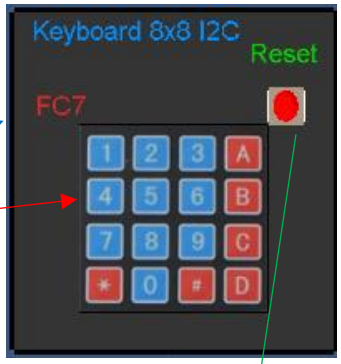
Please use and view the projects in conjunction with this guide. It will help to answer any question that are not given here.

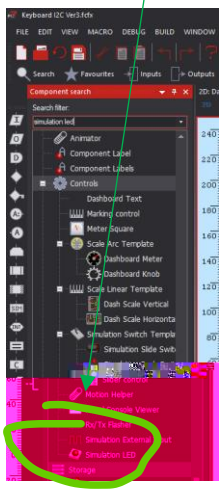
STEP 5 – The Keyboard Components Graphics

The component graphics are built up using simple shapes available under the tools menu. Here is a list of what I have used to construct this keyboard graphic.



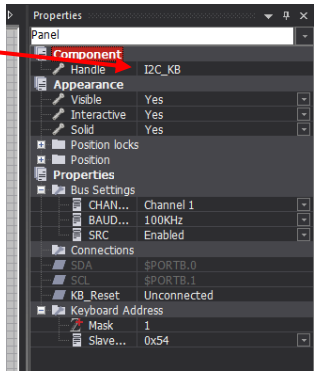
- Rectangles x 2
- Label x 3
- Simulation LED x 1
- Image of 4x4 keypad placed onto second rectangle





STEP 5 – Export Keyboard Components settings

Handle)



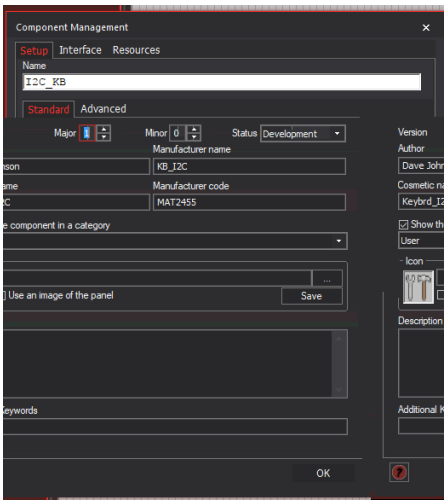
This will be the name given to the component during export.

Start by placing two rectangles onto the dashboard Panel. The largest one to place the labels and LED. And another one just big enough to place the jpeg keypad image. This is necessary as the image will be forced to fill the rectangle holder it is placed onto.

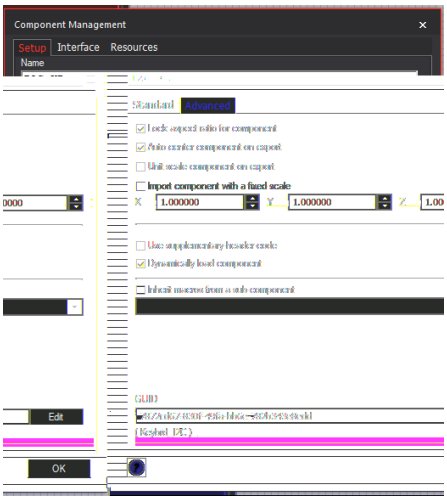
The labels are a matter of taste.

From the File menu open the ***Component Configuration.***

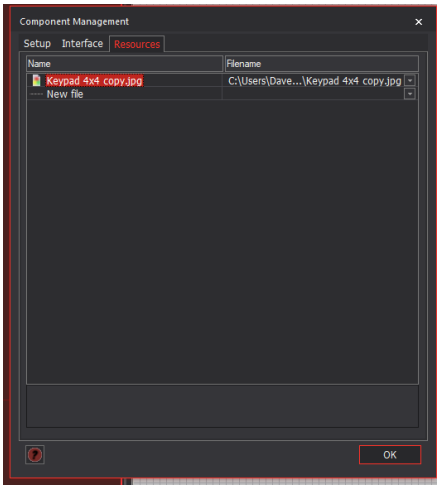
This brings up the Component Management Interface. Complete the field entries to suit your requirements for **Standard**



The **Advanced** TAB, ensure you give the component a unique **GUID** by selecting edit and NEW.



You may need to add the location of where the image used for the keyboard is located as shown (*route folder for component*).



)

What you build yourself, you now know how it works!

Appendix A

Keyboard Controller MAT2455-I/P

Part	Value	Farnell Order No.
C1	100nF	9411887
C2	22pf	9411674
C3	22pf	9411674
JP1	COL_con1_6	Pinhead connectors
JP2	ROW_con1_7	
JP3	PWR 1X02	
JP4	I2C_Con 1X03	
Q1	16Mhz HC49U-V	1611761
R1	10k	9342419
R2	10k	
R5	10k	
R6	10k	
R7	10k	
R8	10k	
R16	4.7k	9343253
R17	2.2k	9342834
T1	2N3904 TO92	
U1	PIC18F2455-I/SP	1579600

Appendix B

A Component can be *anything* that is self-contained, for example:

- An electronic device
- A measuring instrument
- A packaged simulation
- An extension to the system
- A library of useful macros

The Flowcode 7 component contains all the macros in one package. It will enable your project to communicate with a device. Some macros can simulation an activity such as communicating with the device.