ShieldBuddy TC275 User Manual

Basic information on the ShieldBuddy TC275 development board

Connectors, board layout, component placement, power options, programming

PRELIMINARY

User Manual
4269.40100, V1.4, 2015-05

User Manual

Edition 2015-05

Published by:
Hitex (U.K.) Limited.
University Of Warwick Science Park, Coventry, CV4 7EZ, UK

Information

For further information on technology, delivery terms and conditions and prices, please contact the nearest Hitex Office (www.hitex.co.uk).

Document Change History

Date	**Version**	**Changed By**	**Change Description**
8/8/2014	0.1	M Beach	First version
9/8/2014	0.2	M Beach	Revised top view
20/2/2015	0.3	M Beach/D Greenhill	Revised for Rev B HW
9/4/2015	0.8	M Beach	Added board test
16/9/2015	0.9	M Beach	Corrected P33.6
8/11/2016	1.0	M Beach	Added IDE extensions
29/11/2016	1.1	M Beach	Added new connector diagram
9/1/2017	1.2	M Beach	Changed Fast_digitalWrite()
13/1/2017	1.3	M Beach	Added Wire changes
23/1/2017	1.4	M Beach	Added EEPROM support.

We Listen to Your Comments
Is there any information in this document that you feel is wrong, unclear or missing?
Your feedback will help us to continuously improve the quality of this document.
Please send your comments (including a reference to this document) to:
comments@hitex.co.uk

Table of Contents

List of Figures

List of Tables

1 Getting Started

1.1 What Are The ShieldBuddy TC275 Tools?

The main ShieldBuddy toolchain is the Eclipse-based “FreeEntryToolchain” from Hightec/PLS/ Infineon. This is a full C/C++ development environment with source-level debugger. The familiar Arduino IDE is also available for the ShieldBuddy. Both IDEs are based on the Infineon iLLD libraries and allow the usual Arduino C++- like Processing language to be used with the familiar Arduino IO funcitions e.g. digitalWrite(), analogRead(), Serial.print() etc.. These functions are implemented for all three TC275 cores and can be used without restriction.

Given the awesome power of the TC275 we expect most users to program it in C in Eclipse, using the iLLD API directly or working with the underlying SFRs. The neat thing about the ShieldBuddy is that it lets you access the massive power of the TC275 without knowing anything about the bits and bytes of the peripherals!

1.2 Getting Started With The TC275 Toolchain

If you have never used an Arduino-style board before then is a good idea to have a look at www.arduino.cc to find out what it is all about! Although the ShieldBuddy contains three powerful 32-bit, 200MHz processors, it can be used in exactly the same way as an ordinary Arduino Uno. The same Arduino IDE can be used but with an add-on to allow triple core operation. To use the ShieldBuddy you will need:

(i) a PC with Windows Vista or later

(ii) The Aurix free toolchain with Eclipse, C/C++ compiler and UDE debugger from PLS:

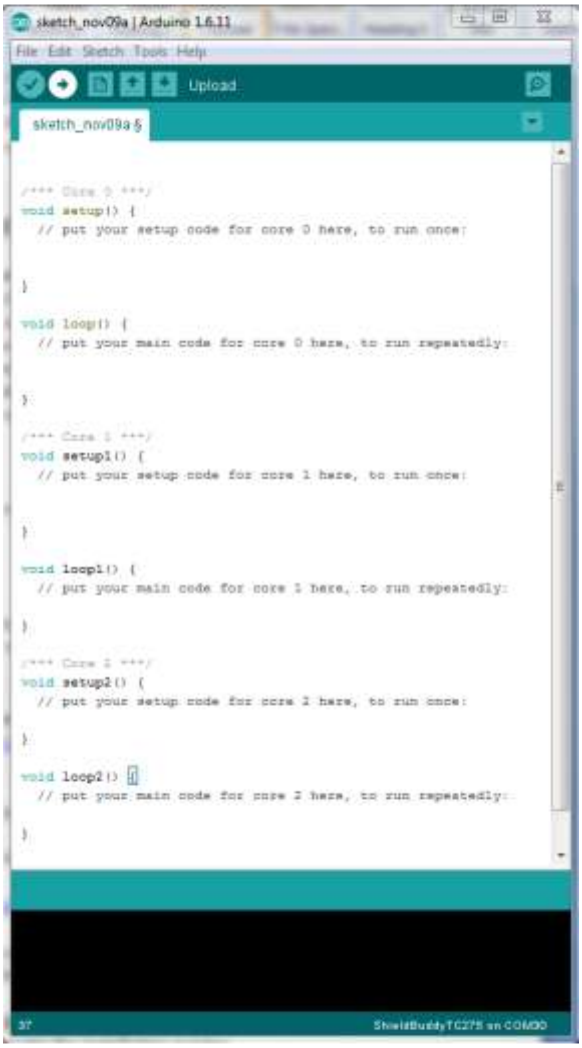
http://free-entry-toolchain.hightec-rt.com/

Follow the instructions given as you will need a free licence file which will be automatically emailed to you. You will need to copy it to : C:\HIGHTEC\licenses.

(iii) The standard 1.6.13 Arduino IDE installed from:
http://arduino.cc/download.php?f=/arduino-1.6.13-windows.exe

Make sure you install this in the default directory!

If you do not have admin rights on your PC then use this version:
https://downloads.arduino.cc/arduino-1.6.13-windows.zip



(iv) The Arduino development environment add-in for Eclipse and the standard Arduino IDE:

http://www.hitex.co.uk/fileadmin/uk-files/downloads/ShieldBuddy/ShieldBuddyMulticoreIDE.zip
Unzip this to a temporary directory using the zip password “ShieldBuddy”. Run the installer and use the password “ShieldBuddy” to copy the IDE onto your PC.

Install these in the order Aurix freetoolchain, Arduino IDE, ShieldBuddy IDE. We hope to combine these into a single installer in the near future to make the installation quicker.

1.3 Using The ShieldBuddy TC275

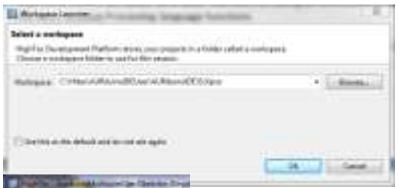
Once all of the above packages have been installed, use the ShieldBuddy just like any other Arduino except that you have three processors to play with rather than just one. Processor core 0 will run setup() and loop() with processor cores 1 and 2 running setup1()/setup2() and loop1()/loop2(). There are no special measures required to write triple-core programs but make sure that that you do not try to use the same peripheral with two different cores at the same time. Whilst nothing nasty will happen, your programs will probably just not work properly! Each core is basically identical except that cores 1 and 2 are about 20% faster than core 0, having an extra pipeline stage. They all can use the same Arduino Processing language functions.

Remember to press the reset button on the ShieldBuddy to make it run the new program.

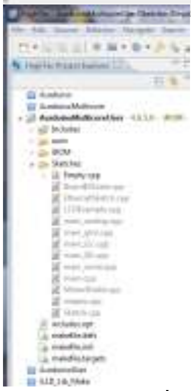
1.4 Using The Eclipse IDE

If you want to use the Eclipse environment, start the toolchain with the  icon. When prompted, open the workspace at:

C:\Hitex\AURDuinoIDE\Eclipse



The default project is AURduinoMulticoreUser:



Arduino-style sketches are stored in the Sketches directory. The default sketch “Empty.cpp” is a simple program that uses all three cores. You can overwrite the statements we used with your own.

To get your programs into the ShieldBuddy, use the PLS  UDE debugger

Open the workspace
“C:\Hitex\ShieldBuddyIDE\Eclipse\ShieldBuddyMulticoreUser\.ude\UDEDefEclipseWorkspace.wsx.”

The program will automatically load. You can run it by clicking the  icon and stop it with the  icon. To reset the program, use the  icon. You can find more information on using the Eclipse tools and the PLS UDE debugger in the guide supplied with the FreeToolChain.

1.5 Getting Help

If you need help, there is a new on-line forum at http://ShieldBuddy.boards.net/.
The hardware user manual with the pinouts is at http://www.hitex.co.uk/index.php?id=3650.

2 ShieldBuddy TC275 Extensions To The Arduino IDE

2.1 How is the ShieldBuddy Different To Other Arduinos?

Most Arduino-style boards use AVR or ARM/Cortex processors which are fine for basic messing about with micros - these chips are everywhere in consumer gadgets and devices. The ShieldBuddy is different, having the mighty Infineon Aurix TC275 processor. These are normally only to be found in state of the art engine management systems, ABS systems and industrial motor drives in your favourite German automobile. They rarely make it out into the daylight of the normal hobbyist/maker world and to date have only been known to a select few at Bosch, BMW, Audi, Daimler-Benz etc..

The standard Arduino IDE can be used, provided that the ShieldBuddy add-in has been installed. Programs can be written in exactly the same way as on an ordinary Arduino. However to make best use of the multicore TC275 processor, there are some special macros and functions available.

2.2 TC275 Processor Architecture

Unlike the AVR, SAM3 etc. used on normal Arduinos, the TC275 has three near-identical 200MHz 32-bit CPU cores on a shared bus, each with their own local RAM but sharing a common FLASH ROM. The peripherals (timers, port pins, Ethernet, serial ports etc.) are also shared, with each core having full access to any peripheral.

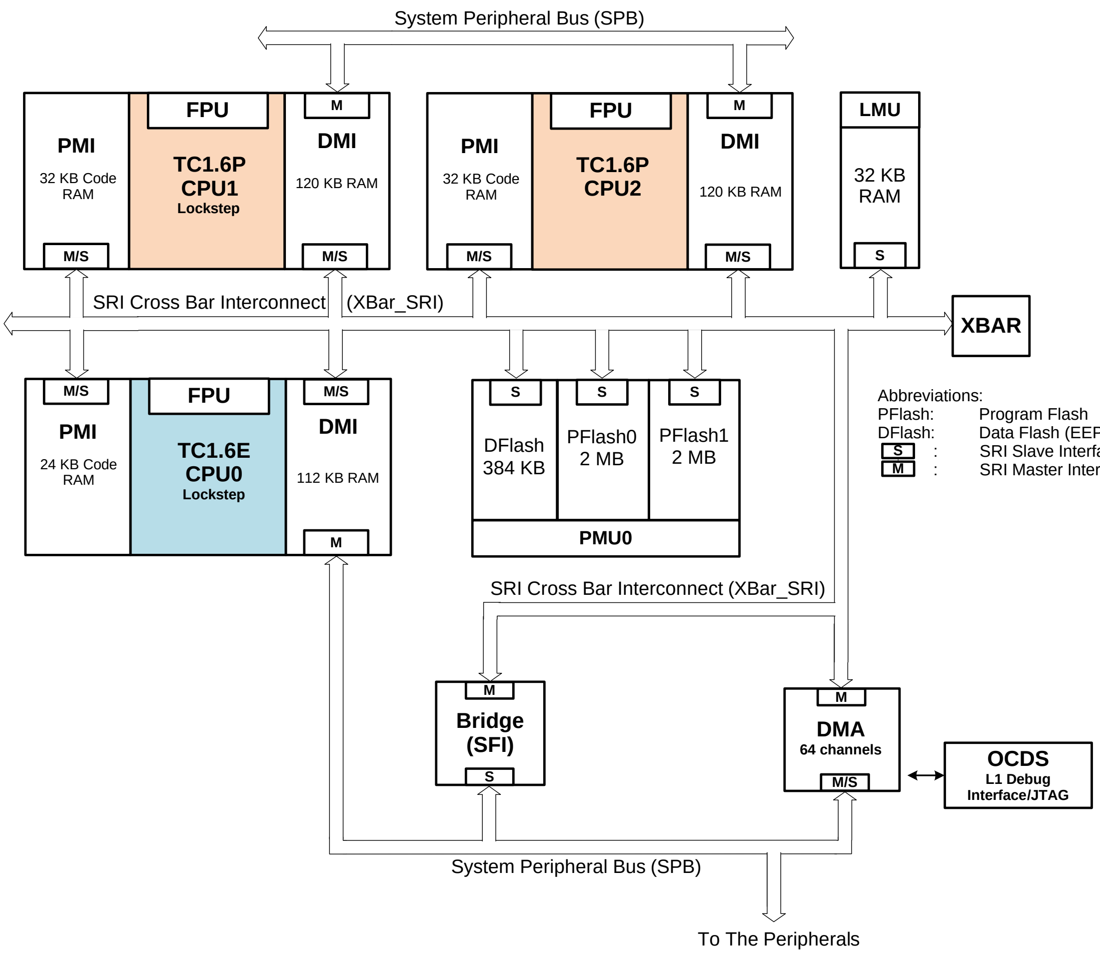


Figure 1 TC275 Internal Layout

The TC275 CPU core design has a basic 5ns cycle time which means you can get typically around 150 to 200 32-bit instructions per microsecond. This is seriously fast when you consider that the Arduino Uno's Atmega328P only manages around sixteen 8-bit instructions/us! In addition, there is a floating point unit on each core so using floating point variables does not slow things down significantly.

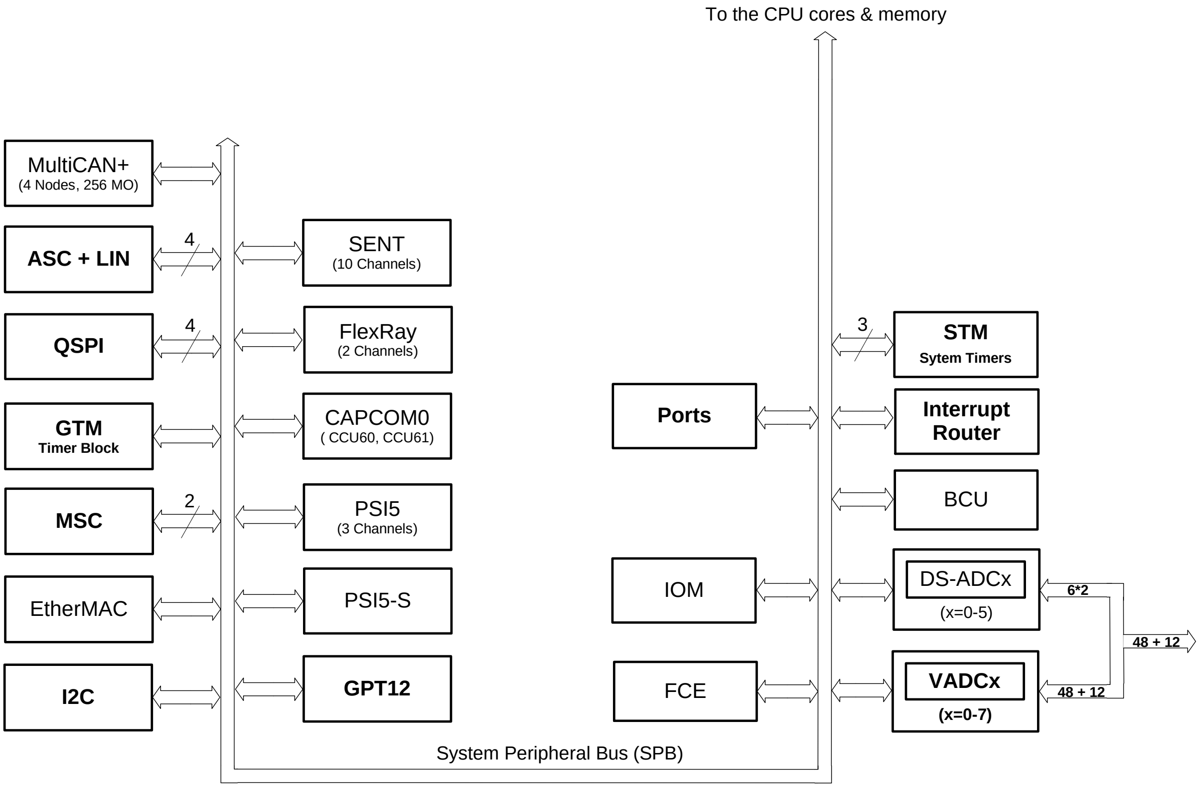


Figure 2 TC275 Peripherals

With so much computing horsepower available, the TC275 can manage a huge range of peripherals. Besides commonplace peripherals like CAN, ADC, I2C, Ethernet, SPI etc. the TC275 has possibly the most powerful signal measurement and generation block to be found on any microcontroller (GTM) plus a an advanced super-fast delta-sigma analog to digital converter.

The Generic Timer Module (GTM) is the main source of pulse generation and measurement functions containing over 200 IO channels. It is designed primarily for automotive powertrain control and electric motor drives. Unlike conventional timer blocks, time-processing units, CAPCOM units etc. it can work in both the time and angle domains without restriction. This is particularly useful for mechanical control systems, switch-reluctance motor commutation, crankshaft synchronisation etc.

Under the bonnet the GTM has around 3000 SFRs but fortunately you do not need to know any of these to realize useful functions! It is enormously powerful and the culmination of 25 years of meeting the needs of high-end automotive control systems. However it can and indeed has been successfully applied to more general industrial applications, particularly in the field of motor control where is can drive up to 4 three-phase motors. The Arduino analogWrite() function makes use of it in a simple way to generate PWM. It can also flash a LED. There is a second timer block (GPT12) can be used for encoder interfaces. Usefully most port pins can generate direct interrupts.

With 176 pins required to get these peripherals out and only 100 pins on the Arduino Due form factor, some functions have had to be limited. The 32 ADC channels have been limited to 12 and the 48 potential PWM channels are also limited to 12, although more channels can be found on the double row expansion connector, if needed.

2.3 Serial Ports

The Arduino has the Serial class for sending data to the UART which ultimately ends up as a COM port on the host PC. The ShieldBuddy has 4 potential hardware serial ports so there are now 4 Serial classes. The default Serial class that is directed to the Arduino IDE Serial Monitor tool becomes SerialASC on the ShieldBuddy. Thus Serial.begin(9600) becomes SerialASC.begin(9600) and Serial.print(“Hi”) becomes SerialASC.print(“Hi”) and so on.

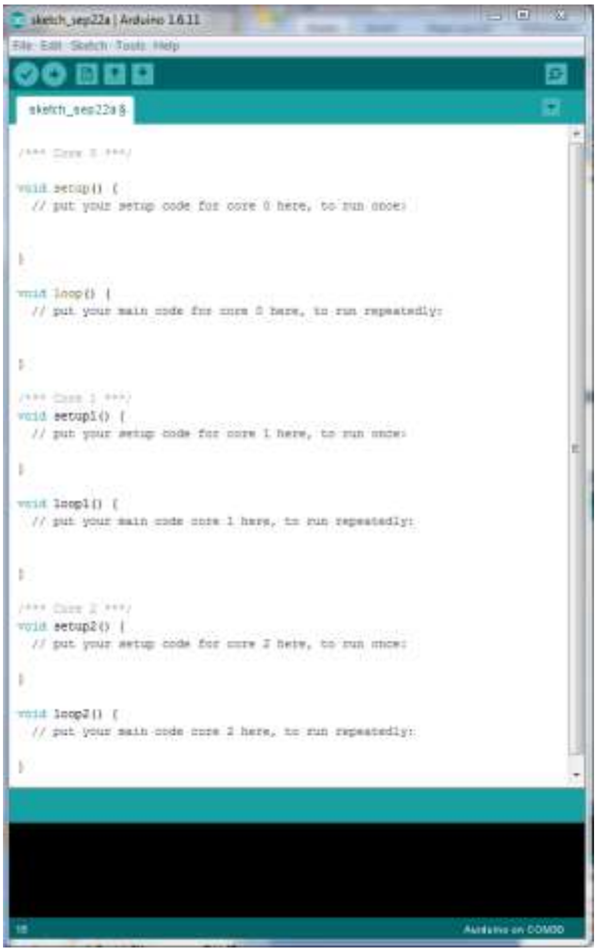
The serial channels are allocated as per:

SerialASC	Arduino FDTI USB-COM	micro USB	
Serial1	RX1/TX1 Arduino	J403	pins 17/16
Serial0	RX0/TX0 Arduino	J403	pins 15/14
Serial	RX/TX Arduino default	J402	pins D0/D1

Any of the serial channels can be used from any core but it is not a good idea to access the same serial port from more than one core at the same time – see the later section on multicore programming.

2.4 Multicore Programming Extensions

2.4.1 Arduino IDE Extensions



The standard Arduino IDE has been extended to allow the all 3 cores to be used. Anybody used to the default Arduino sketch might notice though that in addition to the familiar setup() and loop(), there is now a setup1(), loop1() and setup2(), loop2(). These new functions are for CPU cores 1 and 2 respectively. So while Core0 can be used as on any ordinary Arduino, the lucky programmer can now run three applications simultaneously.

Core0 can be regarded as the master core in the context of the Arduino as it has to launch the other two cores and then do all the initialisation of the Arduino IO, timer tick (for millis() and micros() and delay()). Thus setup1() and setup2() are reached before setup()!

Although all three cores are notionally the same, in fact cores1 and 2 are about 25% faster than core0 as they have an extra pipeline stage. Thus it is usually best to put any heavyweight number crunching tasks on these cores.

Writing for a multicore processor can be a bit mind-bending at first! The first thing to realise is that there is only one ROM and the Arduino IDE just compiles source code. It has no idea (and does not need to know) which core a particular function will run on. It is only when the program runs that this becomes fixed. Any function called from setup and loop() will run on core0; any called from setup1() and loop1() will execute on core1 and so on. Thus is perfectly possible for the same function you wrote to execute simultaneously on all three cores. As there is only one image of this function in the FLASH, the internal bus structure of the Aurix allows all three cores to access the same instructions at the same addresses (worst case) at exactly the same time. Note that if this extreme case happens, there will be a slight loss of performance.

Sharing of functions between cores is easy, provided that they do not make use of the peripherals! Whilst there are three cores, there are only two ADCs. If all three cores want to access the same result register, there is no particular problem with this. However if you want a timer to generate an interrupt and call a shared function, then that function might need to know which core it is currently running on! This is easy to do as there is a macro defined to return the core number.

```
if(GetCpuCoreID() == 2)
{
      /* We must be running on core 2! */
}
```

Fortunately it is rare to have to do this but it is used extensively in the ShieldBuddy to Arduino translation layer.

2.4.2 Inter-Core Communications

One of the aims of the AURIX multicore design is to avoid the awkward programming issues that can arise in multicore processors and make the system architect's job easier. The three independent cores exist within a single memory space (0x00000000 – 0xFFFFFFFF), so they are all able to access any address without restriction. This includes all the peripherals and importantly all FLASH and RAM areas.

Having a consistent global address space when accessing RAM can considerably ease the passing of data between cores using shared structures. Supporting high performance when doing this is achieved by the implementation of a crossbar bus system to connect the cores, memories and DMA systems. Of course there are protection mechanisms that can generate traps for such accesses if the application requires it, as they may indicate a program malfunction which would need to be handled in an orderly manner.

The upshot of this is that the programmer does not need to worry about cores accessing the same memory location (i.e. variable) at the same time. In some multicore processors this would cause an exception and is regarded as an error. Certainly if you are new to multicore programming, this makes life much easier. Of course there could be a contention at the lowest level and this can result in extra cycles being inserted but given the speed of the CPU, this is unlikely to be an issue with Arduino-style applications.

With an application split across three cores, the immediate problem is how to synchronise operations. As the Aurix design allows RAM locations to be accessed by any core at any time, this is no problem. In the simplest case, global variables can be used to allow one core to send a signal to another. Here is an example.

2.4.2.1 Inter-Core Communications Example

We want to use the SerialASC.print() function to allow each core to send a message to the Arduino Serial Monitor – something like “Hello From Core 0”, “Hello From Core 1” etc..

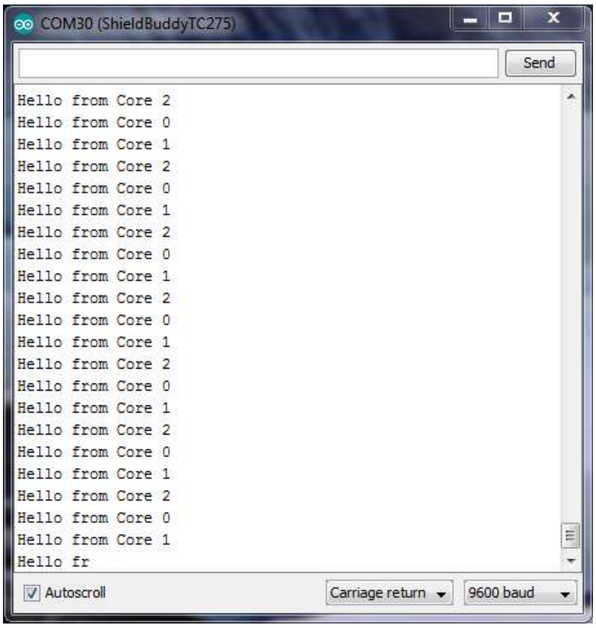
If we do nothing clever and just allow each core's Loop() to write to the SerialASC, we get a complete jumble of characters. This is because each core will write to the transmit buffer at random times. The Aurix does not care that 3 cores are trying to use the same serial port and nothing nasty like an exception will happen. All the characters are in there from all the cores but not necessarily in the right order.

What we need to do is make sure that each core waits in turn for the other cores to finish writing to the serial port. This is quite easy using some global variables. However with true multicore programming, weird things can happen that don't occur in single core.

An obvious approach to solving this is to have a global variable that tells everybody whether the SerialASC port is being used. However this does not work where we are trying to prevent a single resource (e.g. serial port) being simultaneously accessed from two cores. It can work where we simply want to pass variables between cores though. The problem is that other cores can do anything at any time relative to each other. If Cores1 and 2 both execute the check of the SerialASCInUse flag at around the same time, they will both see it as ‘0’ and then both set it to ‘1’. In practice it is when Core2 checks the flag in the few instructions between Core1 checking it for ‘0’ and then setting it to ‘1’, that we get into trouble. They will then both attempt to write to the SerialASC port, with the result that garbage gets sent to the terminal.

To solve this tricky problem, we need a means of checking the SerialASCInUse flag for ‘0’ and setting it to ‘1’ in a single Aurix instruction. This means that there would be no gap within which another core could get it. This is catered for by the uint32 Htx_LockResource(uint32 *ResourcePtr) function. This sets the flag at address ResourcePtr automatically to Htx_RESOURCE_BUSY = 1 and returns the previous flag state.

The ShieldBuddy serial port classes have been extended by adding a “PortInUse” variable so that multicore support is now built in. Using the Htx_LockResource() function, we can ensure that no two cores will try to access the SerialASC at the same time.



This is rather inefficient way of getting cores to work together as the cores spend a lot of time hanging around in while() loops. Another way is to get one core to create an interrupt in another core to tell it to do something.

2.4.2.2 Using Interrupts To Coordinate and Communicate Between Cores.

The Arduino language has been extended to allow you to trigger an interrupt in another core. This means that core 0 can trigger an interrupt in say core 1. That interrupt might tell Core 0 that a resource is now free or perhaps tell it to go and read a global variable that core0 has just updated.

```
/* Create an interrupt in core 1 */
CreateCore1Interrupt(Core1IntService);
```

Here Core1IntService is a function written by the user that Core 1 will execute when Core 0 requests it to do so.

Here is an example of coordinating the three cores to use the SerialASC port again. Now the print to the SerialASC port only takes place when (in this example) core0 requests it.

There are three CreateCoreXInterrupt() functions available, one for each core. The parameter passed is the address of the function that you want to run in the other core:

```
/* Create an interrupt in core 0 */
CreateCore0Interrupt(Core0IntService);

/* Create an interrupt in core 1 */
CreateCore1Interrupt(Core1IntService);

/* Create an interrupt in core 2 */
CreateCore2Interrupt(Core2IntService);
```

These can be used with any core (i.e. in setup(), setup1() and setup2()). Thus any core can run an interrupt fuction in any other core. To trigger the interrupt to happen, the InterruptCoreX() function is used.

```
/* Trigger interrupt in Core0 now! */
InterruptCore0();

/* Trigger interrupt in Core1 now! */
InterruptCore1();

/* Trigger interrupt in Core2 now! */
InterruptCore2();
```

2.4.3 Timers/Ticks/delay(), millis(), micros() Etc.

The TC275 STM0 (system timer 0) is used to as a basis for all the Arduino timing functions such as delay(), millis(), micros() etc. This is based on a 10ns tick time. In addition, the user can create his own timer-based interrupts in core 0 using the CreateTimerInterrupt() function.

This is used as per:

```
void STM0_inttest(void)
{
      digitalWrite(2, ToggleVar0 ^= 1);
}

void setup() {

      /* 10ns per bit count */
      CreateTimerInterrupt(ContinuousTimerInterrupt, 10000, STM0_inttest);
```

Here the user wants his function “STM0_inttest() to run every 100us forever. The time is specified in units of 10ns so 100us =10000 * 0.01us. For 50us, the value would be 5000. This can be used for making simple task schedulers.

If the STM0_inittest() is only intended to run once but in 100us from now, this would be used:

```
/* Run STM0_inttest once, 100us in the future */
CreateTimerInterrupt(OneShotTimerInterrupt, 10000, STM0_inttest);
```

The maximum time period that can be set is about 42 seconds. The minimum practical time period is around 20us. If you want something faster then you will need to use another method!

For cores 1 and 2, there are further timer interrupt creation functions, using STM1 and STM2. There are two timer interrupts per core allowed using this method (other methods allow more!).

2.4.3.1 Core 1

The CreateTimerInterrupt0_Core1() function and CreateTimerInterrupt1_Core1() allow two independent interrupt functions to be called freely in the same way as with core0's CreateTimerInterrupt(). These use STM1.

For example:

```
void STM1_inttest0(void)
{
   digitalWrite(3, ToggleVar1 ^= 1);
}

void STM1_inttest1(void)
{
   digitalWrite(4, ToggleVar2 ^= 1);
}

/* Make STM1_inttest0() function run every 100us */
CreateTimerInterrupt0_Core1(ContinuousTimerInterrupt, 10000, STM1_inttest0);

/* Make STM1_inttest1() function run every 50us */
CreateTimerInterrupt1_Core1(ContinuousTimerInterrupt, 5000, STM1_inttest1);
```

2.4.3.2 Core 2

For Core2 there are similar functions to core 1 but which are now based on STM2:

```
void STM2_inttest0(void)
{
   digitalWrite(5, ToggleVar3 ^= 1);
}

void STM2_inttest1(void)
{
   digitalWrite(6, ToggleVar4 ^= 1);
}

/* Make STM2_inttest0() function run every 100us */
CreateTimerInterrupt0_Core2(ContinuousTimerInterrupt, 10000, STM2_inttest0);

/* Make STM2_inttest1() function run every 50us */
CreateTimerInterrupt1_Core2(ContinuousTimerInterrupt, 5000, STM2_inttest1);
```

2.4.3.3 Direct Fast Access To The System Timer0

To read the current value of the STM0, upon which all the timing functions are based, use the GetCurrentNanoSecs() function. This returns the current timer value in steps of 10ns.

```
TimeSnapshot0 = GetCurrentNanoSecs();
for(i = 0; i < 500; i++)
{ ; }
TimeSnapshot1 = GetCurrentNanoSecs();
/* Time in units of 10ns */
ExecutionTime = TimeSnapshot1 - TimeSnapshot0;
```

2.4.4 Managing the Multicore Memory Map

The Arduino IDE gives no clue as to where anything goes or even what memory is available. If you are not bothered about execution speed or are only using Core 0, then variables can be declared just as in any other Arduino board. However if you are using Cores1 & 2, having some idea how the physical memory is arranged inside the TC275 can make a huge difference to the maximum performance that can be obtained.

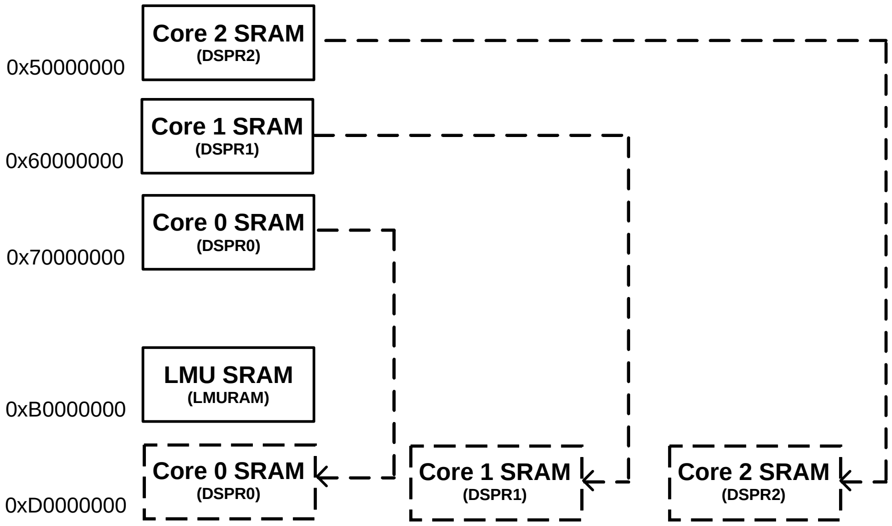


Figure 3 TC25 RAMs

A global variable declared in the usual way will end up in the Core 0 SRAM ("DPSR0").

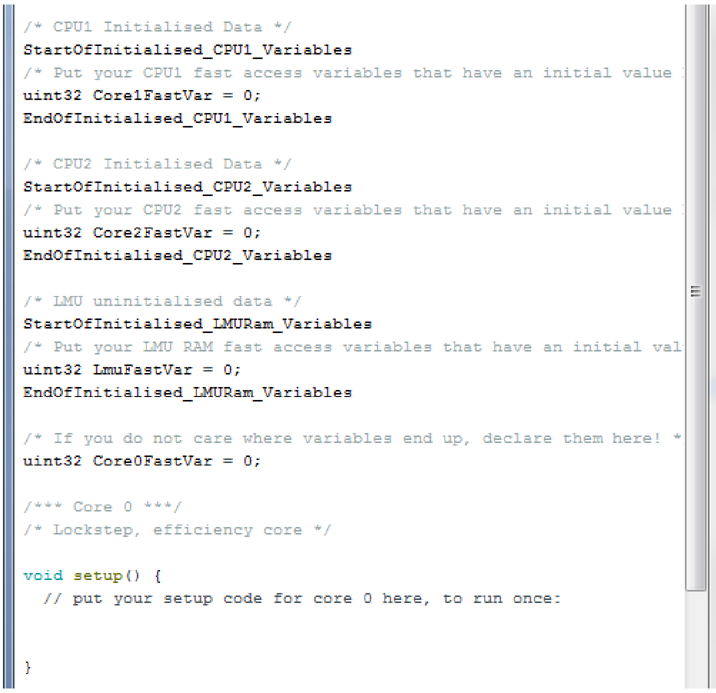
```
/* If you do not care where variables end up, declare them here! */
uint32 myglobalvariable = 0;

/*** Core 0 ***/
/* Lockstep, efficiency core */

void setup() {
  // put your setup code for core 0 here, to run once:


}
```

If this is only used by Core0 then the access time will be very fast. This is because each of the RAMs appears at two addresses in the memory map. Core0's DSPR RAM appears to be at 0xD0000000 where it is considered to be local and is directly on Core0's local internal bus. It is also visible to the other cores at 0x70000000 so that they can read and write it freely. The penalty is that the access will be via a bus system that all cores can access (the SRI) which unfortunately is much slower and can be influenced by other traffic between cores. Thus all the cores have local RAM that is visible to the other cores, albeit at reduced speed.

There is a fourth RAM area ("LMU") which is not tied directly to any core and which all cores have fast access to. This is useful for shared variables that are heavily used by all cores.

As cores 1 & 2 are the fast cores, it makes sense to put their variables into their local RAMs but as standard, the Ardunio IDE has no support for this. For the ShieldBuddy, a series of ready-made macros are available that allow you to put variables into any of these SRAM areas easily.

Using these macros for core 1 and 2 data will give a significant increase in performance and is highly recommended.



The complete set of macros for putting variables in specific RAMs is:

```
/* LMU uninitialised data */
StartOfUninitialised_LMURam_Variables
/* Put your LMU RAM fast access variables that have no initial values here e.g. uint32 LMU_var; */
EndOfUninitialised_LMURam_Variables

/* LMU uninitialised data */
StartOfInitialised_LMURam_Variables
/* Put your LMU RAM fast access variables that have an initial value here e.g. uint32 LMU_var_init = 1;
*/
EndOfInitialised_LMURam_Variables

/* CPU1 Uninitialised Data */
StartOfUninitialised_CPU1_Variables
/* Put your CPU1 fast access variables that have no initial values here e.g. uint32 CPU1_var; */
EndOfUninitialised_CPU1_Variables

/* CPU1 Initialised Data */
StartOfInitialised_CPU1_Variables
/* Put your CPU1 fast access variables that have an initial value here e.g. uint32 CPU1_var_init = 1;
*/
EndOfInitialised_CPU1_Variables

/* CPU2 Uninitialised Data */
StartOfUninitialised_CPU2_Variables
/* Put your CPU2 fast access variables that have no initial values here e.g. uint32 CPU2_var; */
EndOfUninitialised_CPU2_Variables

/* CPU2 Initialised Data */
StartOfInitialised_CPU2_Variables
/* Put your CPU2 fast access variables that have an initial value here e.g. uint32 CPU2_var_init = 1;
*/
EndOfInitialised_CPU2_Variables
```

2.5 Peripheral And IO Extensions

2.5.1 Faster digitalRead & digitalWrite

These functions are identical to the Arduino versions but to some extent suffer from limited performance due to the overhead of the Arduino hardware abstraction layer.

Example of writing to Pin 2.

```
digitalWrite(2,HIGH);  // 160ns, 6.25MHz core 0, 120ns core1/2
digitalWrite(2,LOW);
```

The maximum pin toggling rate is 6.25MHz on core 0 and 8.3MHz on cores 1 &2

To allow a more direct access to the IO pins, the Fast_digitalWrite() is provided.

```
Fast_digitalWrite(2, LOW);  // 30ns, 25MHz Core1/2, 40ns, core0
Fast_digitalWrite(2, HIGH);
```

The run time is shorter, allowing the pin to be toggled at up to 25MHz when using cores 1 and 2.

The Fast_digitalRead(2) is a faster equivalent to digitalRead(2).

2.5.2 attachInterrupt() Function

The Arduino attachInterrupt() function is supported with some minor differences. The following pins are able to create interrupts:

2, 3, 15, 18, 20, 52

The mode parameter supports only values of RISING, FALLING, CHANGE.

ASC and QSPI are still available from functions called from these interrupts but timer functions created from the CreateTimerInterrupt() function are not.

2.5.3 ADC Read Resolution

The default resolution for ADC conversion results is 10 bits, like on an ordinary Arduino. On the ShieldBuddy you can set 8-bit or 12-bit conversions if required, using the analogReadResolution () function.

```
/* Set default VADC resolution 10 bits */
analogReadResolution (10u);
```

To set 12-bits of resolution,:

```
analogReadResolution (12u);
```

To set 8-bits:

```
analogReadResolution (8u);
```

2.5.4 Fast ADC Conversions

The normal Arduino means of reading an analog channel is “analogRead(channel_no). This allows up to around 450ksamples/sec. If the analog channel is fixed, then it is possible to access the channel result directly and get up to around 600ksamples/sec. This requires the use of the ReadADx() functions. There are 12 of these, one for each ShieldBuddy analog channel.

Example of ReadADx():

```
VADC_result[i] = ReadAD0();  // Read A0 directly
VADC_result[i] = ReadAD1();  // Read A1 directly
```

2.5.5 AnalogOut

2.5.5.1 PWM Frequency

Like the Arduino, the ShieldBuddy uses PWM to generate analog voltages. The PWM frequency is only around 1kHz on the Arduino. The ShieldBuddy frequency is 390kHz when using 8-bit resolution. Whilst this is great for AC waveform generation, audio applications etc., it can be too high for some power devices used for things like motor control.

The useArduinoPwmFreq() function will set the PWM frequency to 1.5kHz so that motor shields etc. should work without problems.

2.5.5.2 DAC0 and DAC1 pins

These Arduino pins are specifically for accurate digital to analog conversion. They have a fixed 14-bit resolution (0-16383) and a 6.1kHz PWM frequency.

```
analogWrite(DAQ0, 8192); // Set 2.5V on DAC0 pin
analogWrite(DAQ1, 4096); // Set 1.25V on DAC1 pin
```

2.6 CAN

Controller Area Network is supported via the CANRX/CANTX pins and J406 (double row connector) pins 23 and 22. These are CAN0 and CAN1 modules respectively. 11 and 29-bit message IDs can be used. A total of 16 message objects (or more simply, messages) can be used. This is a subset of the TC275's real capability and is limited for the sake of simplicity.

Some prior knowledge of CAN is required to use these functions!

2.6.1 CAN Functions Usage

First the CAN module(s) must be initialised with the required Baudrate:

```
/*** Core 0 ***/

void setup() {
  // put your setup code for core 0 here, to run once:

      CAN0_Init(250000);
      CAN1_Init(250000);
```

Next the messages to be sent or received via CAN must be set up. Here we will setup a transmit message on CAN0 and receive it on CAN1 (we have connected two CAN modules together):

Transmit Message

```
/* Parameters CAN ID, Acceptance mask, data length, */
/* 11 or 29 bit ID, Message object to use           */
CAN0_TxInit(0x100, 0x7FFFFFFFUL, 8, 11, 0);
```

Receive Message

```
/* Parameters CAN ID, Acceptance mask, data length, */
/* 11 or 29 bit ID, Message object to use           */
CAN1_RxInit(0x100, 0x7FFFFFFFUL, 8, 11, 1);
```

Here we are setting up a message object in the CAN0 module (CANRX/CANTX pins) to send 8 bytes with a message ID of 0x100, using 11-bit identifiers. We will be using message object0 for the transmit message. There are a total of 16 message objects available in the ShieldBuddy CAN driver and it is up to the user to make sure that each transmit and receive object has an unique message object number! In our example, if we set up another message (receive or transmit) we will use object 2, as 0 and 1 are already in use.

To send the message on CAN0 with 8 bytes of data consisting of 0x12340000 (lower 4 bytes) and 0x9abc000 (upper 4 bytes) with message ID 0x100:

```
/* Parameters CAN ID, 32 bits low data, 32 bits high data, data length */
CAN0_SendMessage(0x100, 0x12340000, 0x9abc0000, 8);
```

To receive the message on CAN1:

```
/* Parameters CAN ID, address of structure to hold returned data, data length */
RxStatus = CAN1_ReceiveMessage(0x100, &msg1, 8);
```

For the receive message function, we must provide a structure into which the receive function can place the received data. The predefined structure type “IfxMultican” can be used for this:

```
IfxMultican_Message msg1;
```

The data received can be accessed in:

```
LowerData = msg1.data[0];
UpperData = msg1.data[1];
```

The receive function also returns a status value which can help in the event of a message reception failure. The predefined type “IfxMultican_Status” can be used:

```
IfxMultican_Status RxStatus;
```

The return values are any one of:

```
IfxMultican_Status_noError             = 0x00000000
IfxMultican_Status_notInitialised      = 0x00000001
IfxMultican_Status_wrongParam          = 0x00000002
IfxMultican_Status_wrongPin            = 0x00000004
IfxMultican_Status_busHeavy            = 0x00000008
IfxMultican_Status_busOff              = 0x00000010
IfxMultican_Status_notSentBusy         = 0x00000020
IfxMultican_Status_receiveEmpty        = 0x00000040
IfxMultican_Status_messageLost         = 0x00000080
IfxMultican_Status_newDataButOneLost = 0x00000100
```

Please note that the CAN receive function does not need to know which message object in the CAN module is being used – it works it out from the message ID passed to it. However this relies on any message ID only being used once, which is a basic requirement of the CAN specification anyway.

If the CAN receive functions are run but there is no message waiting then they will return value of 0x40.

2.6.1.1 Receiving any message regardless of message ID

If you want to receive all messages on the CAN bus into a single message object, the acceptance mask parameter in the CANx_RxInit() function needs to be set to zero.

/* Receive all message IDs up t0 0xFFF */
CAN1_RxInit(0x200, 0x7FFFF000UL, 8, 11, 1);

Now the CAN message ID can be anything from 1 to 0xFFF so you can enter any otherwise unused and valid 11 or 29-bit ID. Here we used 0x200. To receive the messages, use:

```
RxStatus = CAN1_ReceiveMessage(0x200, &msg1, 8);
```

2.7 I2C/Wire Pins & Baudrate

The ShieldBuddy's I2C is on pins 20 (SDA) and 21 (SCL). Currently only the master mode is supported. There are two new functions available compared with the Arduino. Before calling the Wire.begin(), the pins to be used for the I2C can be specified, along with the Baudrate. The default pins are 20 and 21 but an alternative set are at pins 6 (SDA) and 7 (SCL) as these are used on some shields.

```
Wire.setWirePins(UsePins_20_21); // Default pins for Arduino Due/MEGA
```

Or:

```
Wire.setWirePins(UsePins_6_7);

Wire.begin();         // join i2c bus (address optional for master)
```

The default Baudrate is 100kbit/s but this can be changed to up to 400kbit/s

```
Wire.setWireBaudrate(400000);    // Set high speed mode

Wire.begin();         // join i2c bus (address optional for master)
```

2.8 EEPROM Support

The Arduino EEPROM functions are available but their use is slightly different to when on an Arduino Uno, MEGA, Due etc. This is because the TC275 has DFLASH rather than EEPROM. This has a similar number of write cycles (125k) but due to the 8kbyte sector size, the mechanism for writing is different. There are 8kbytes of emulated EEPROM available to you. Most of the features of the EEPROM system are described at:

https://www.arduino.cc/en/Reference/EEPROM

Note: The total DFLASH size if 384kbyte and if you want to use it with very large data sets then do not use the Arduino-style EEPROM functions!

Date can be written to and read from the emulated EEPROM one byte at a time. If the EEPROM is to be used in an application, it is recommended that the EEPROM manager is initialised before any read or write operations.

```
/* Initialise EEPROM system */
if(EEPROM.eeprom_initialise() == EEPROM_Not_Initialised)
{
      /* EEPROM is bad */
      while(1) { ; }
}
```

It is not mandatory to do this but if there is a failure in the EEPROM then it will not be reported. It is also the case that the first read or write will initialise the EEPROM manager but please note that the first such operation will take several milliseconds and if there is a failure in the EEPROM, you will not know about it.

EEPROM data can be read freely. EEPROM writes can be done freely as in fact the data is captured in a RAM buffer. Once all the writes required by the application are completed, the eeprom_update() function must be used to program the data into the underlying DFLASH.

```
/* Write buffer to DFLASH */
EEPROM.eeprom_update();
```

This should not be confused with the EEPROM.update() function. This only stops data being written into the RAM buffer if the same data is already there.

2.9 Resetting The ShieldBuddy

It is possible to reset the ShieldBuddy by executing the Reset_TC275() function. This causes a TC275 system reset which puts the CPUs into the reset state.

3 Hardware Release Notes HW Revision B

3.1 ShieldBuddy RevB Known Problems

The ShieldBuddy Revision B has a number of functional characteristics, listed below.

1. It will only run at 5V. It is possible to get 3V3 operation but this requires the changing of a voltage regulator and the changing of some resistors

3.2 CIC61508 (Safety version only)

The CIC61508 is programmed with the VANIA3.2 firmware release.

3.3 VIN Pin

The VIN pin on the ShieldBuddy power connector strip allows access to the 9-12V input from the power jack socket. This may be used to power shields that require a higher voltage e.g. the DC motor shield. In this case, please note that the maximum continuous current that can be drawn through this pin is 1.5A due to the 0.5mm track used..

4 Arduino-Based Connector Details

The ShieldBuddy is based on the Arduino Due (SAM3X) form factor. Where possible, the pin functionality of the Due has been implemented using an equivalent Aurix peripheral.

4.1 Board Layout

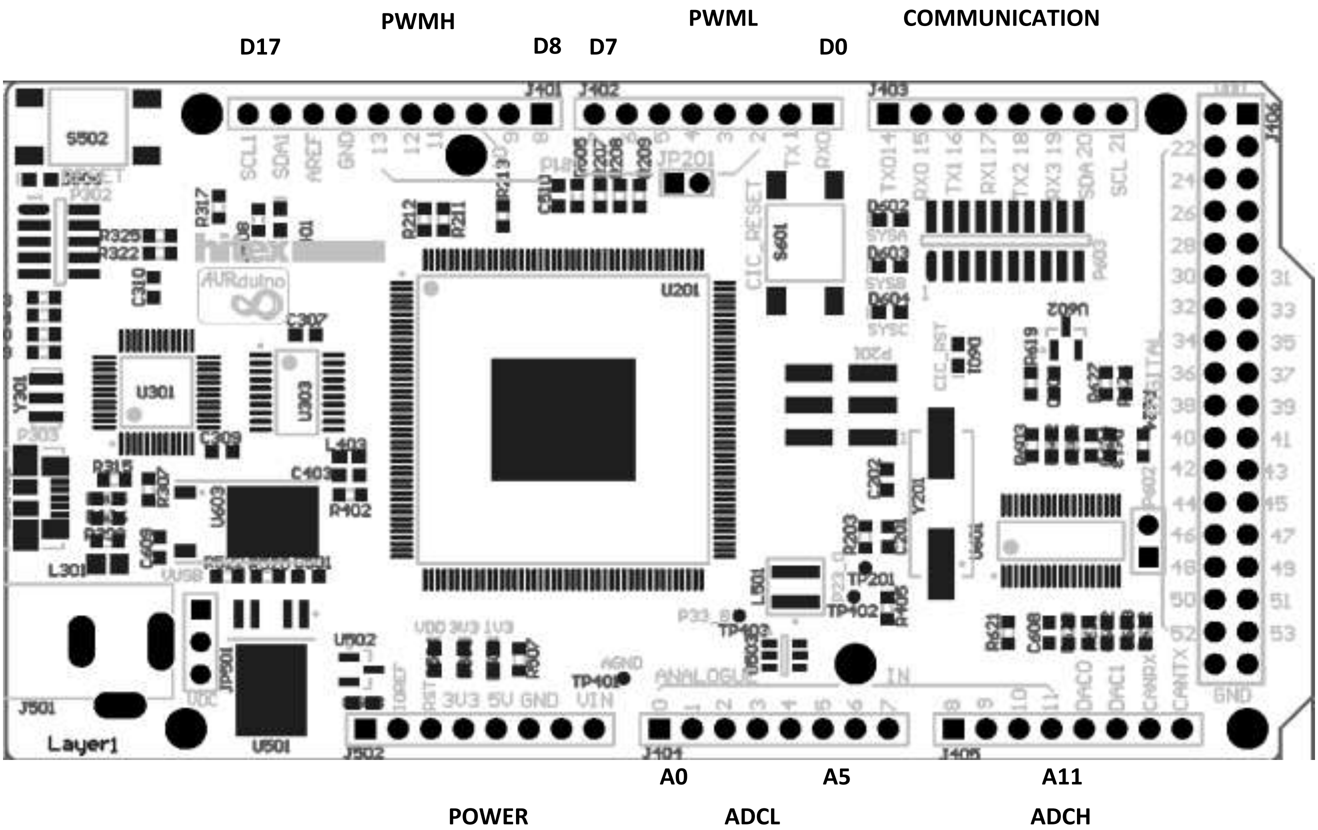
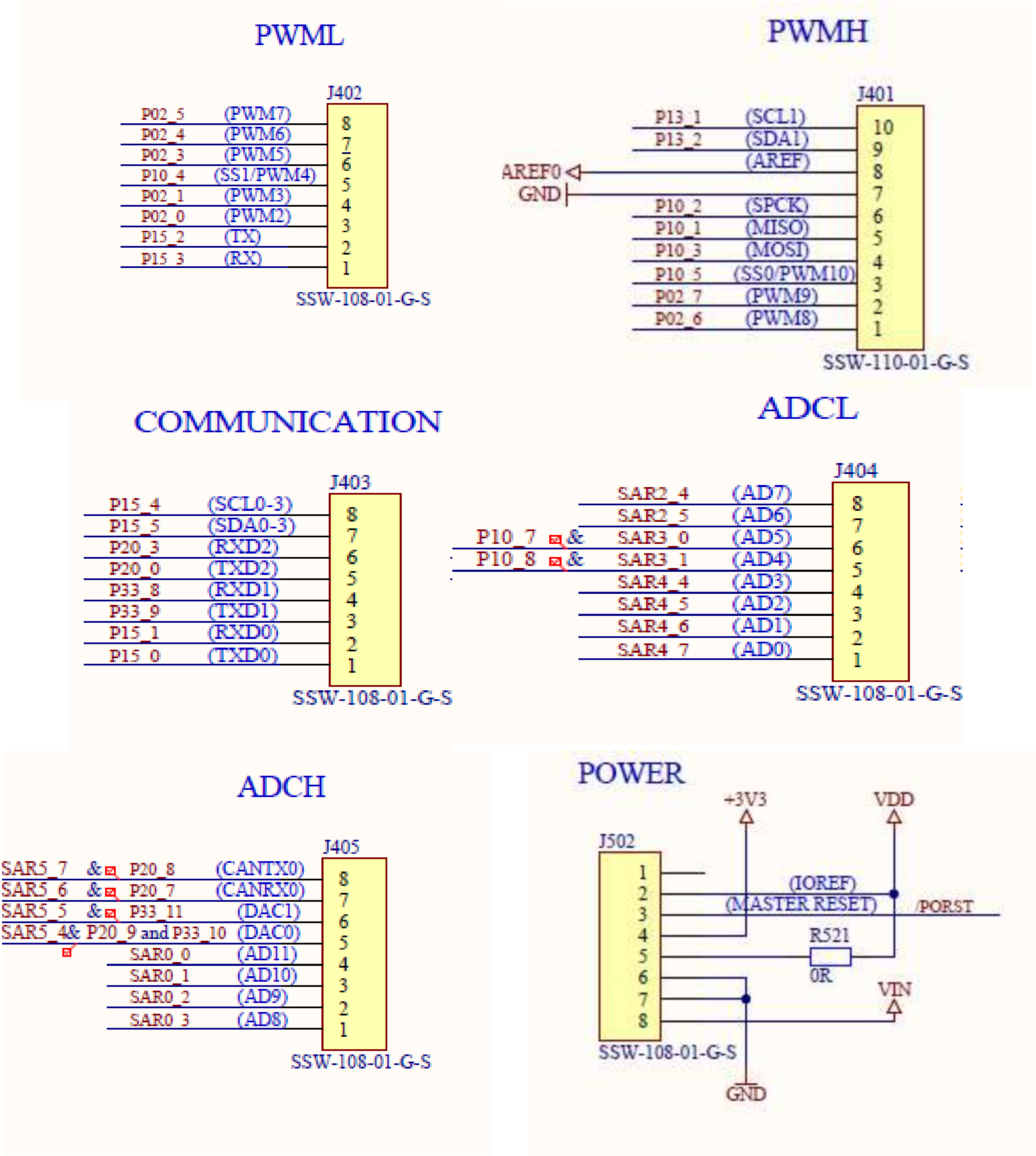


Figure 4 Top view of ShieldBuddy

4.2 Connector Pin Allocation



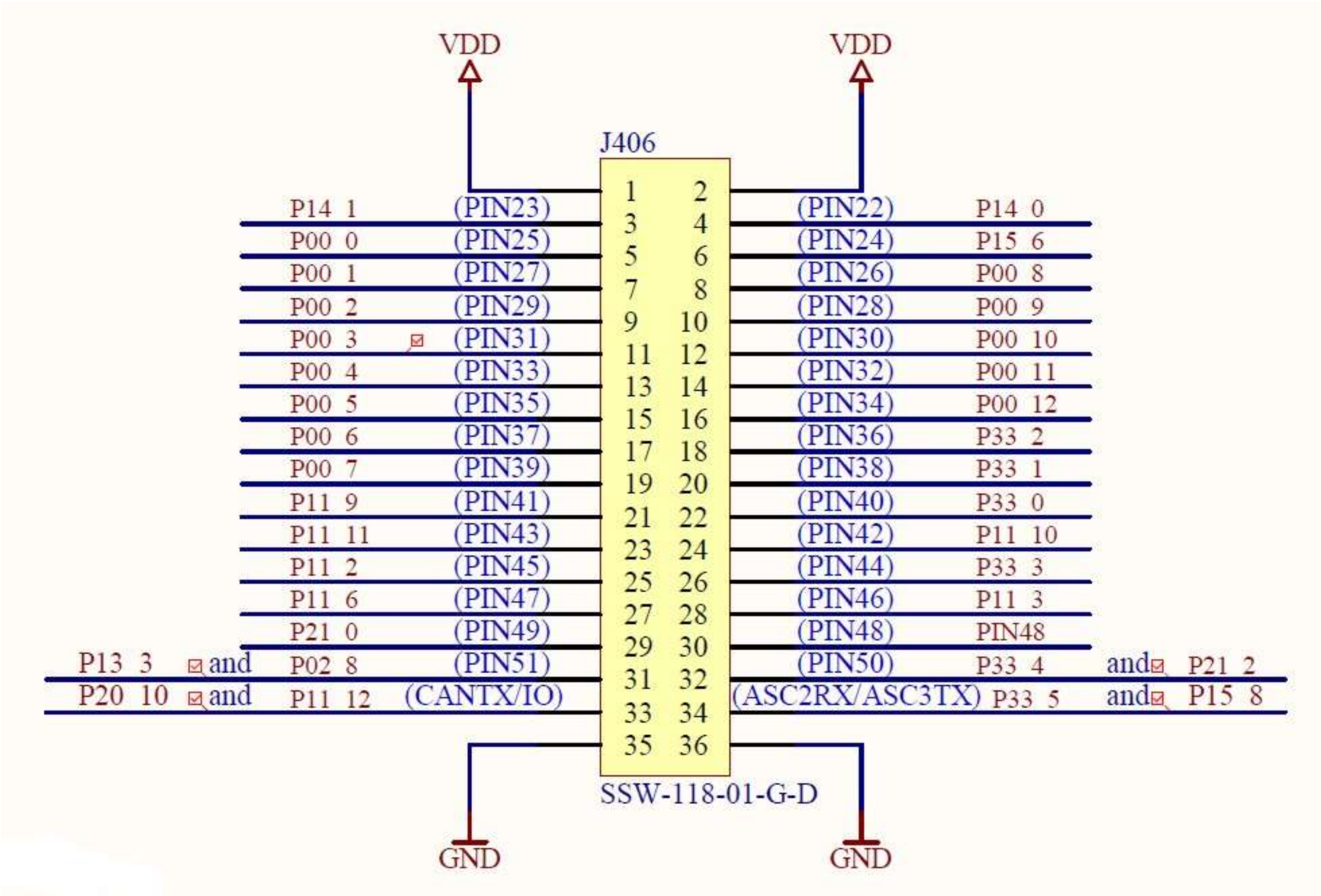


Figure 5 Extended IO Connector

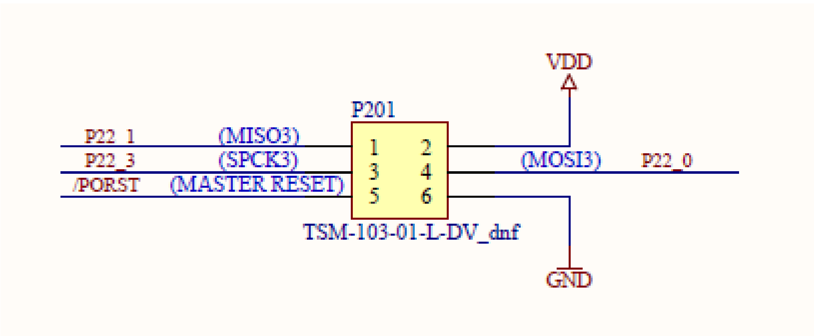


Figure 6 SPI Connector

4.3 TC275 ASCLIN to ShieldBuddy connector mapping

Table 1 ASCLIN to ShieldBuddy connector mapping

TC275 Port Pin	ASCLIN	Board Marking	Comment
P15.0	ASC1	TX0	Serial0
P15.1	ASC1	RX0	
P33.9	ASC2	TX1	Serial1
P33.8	ASC2	RX1	
P20.0	ASC3	TX2	Serial2
P20.3	ASC3	RX2	
P15.2	ASC0	TX	Serial
P15.3	ASC0	RX	
P15.7	ASC3	TX	SerialASC - Available via USB
P32.2	ASC3	RX	

Table 2 Arduino To ShieldBuddy To TC275 Mapping

Arduino Signal Name	ShieldBuddy Connector Name	TC275T Pin Assignment
Analog pin 0	ADCL.1	SAR4.7
Analog pin 1	ADCL.2	SAR4.6
Analog pin 2	ADCL.3	SAR4.5
Analog pin 3	ADCL.4	SAR4.4
Analog pin 4	ADCL.5	SAR3.0/P10.7
Analog pin 5	ADCL.6	SAR3.1/P10.8
Analog pin 6	ADCL.7	SAR2.5
Analog pin 7	ADCL.8	SAR2.4
Analog pin 8	ADCH.1	SAR0.3
Analog pin 9	ADCH.2	SAR0.2
Analog pin 10	ADCH.3	SAR0.1
Analog pin 11	ADCH.4	SAR0.0
Analog pin 12/DAC0	ADCH.5	SAR5.4/P20.9/P33.10
Analog pin 13/DAC1	ADCH.6	SAR5.5/P33.11
Analog pin 14/CAN RX	ADCH.7	SAR5.6/P20.7 CAN0 RX
Analog pin 15/CAN TX	ADCH.8	SAR5.7/P20.8 CAN0 TX
Digital pin 4 (PWM/SS)	PWML.5	P10.4
Analog Reference AREF	PWMH.8	AREF
Digital pin 0 (RX0)	PWML.1	P15.3
Digital pin 1 (TX0)	PWML.2	P15.2
Digital pin 2 (PWM)	PWML.3	P2.0
Digital pin 3 (PWM)	PWML.4	P2.1
Digital pin 5 (PWM)	PWML.6	P2.3
Digital pin 6 (PWM)	PWML.7	P2.4
Digital pin 7 (PWM)	PWML.8	P2.5
Digital pin 8 (PWM)	PWMH.1	P2.6
Digital pin 9 (PWM)	PWMH.2	P2.7
Digital pin 10 (PWM/SS)	PWMH.3	P10.5
Digital pin 11 (PWM/MOSI)	PWMH.4	P10.3
Digital pin 12 (PWM/MISO)	PWMH.5	P10.1
Digital pin 13 (PWM/SPCK)	PWMH.6	P10.2

Arduino Signal Name	ShieldBuddy Connector Name	TC275T Pin Assignment
Digital pin 14 (TX3)	COMMUNICATION.8	P15.0 ASC1 RX TXCAN2
Digital pin 15 (RX3)	COMMUNICATION.7	P15.1 ASC1 RX RXCAN2
Digital pin 16 (TX2)	COMMUNICATION.6	P33.9 ASC2 TX
Digital pin 17 (RX2)	COMMUNICATION.5	P33.8 ASC2 RX
Digital pin 18 (TX1)	COMMUNICATION.4	P20.0 ASC3 TX
Digital pin 19 (RX1)	COMMUNICATION.3	P20.3 ASC3 RX
Digital pin 20 (SDA)	COMMUNICATION.2	P15.4
Digital pin 21 (SCL)	COMMUNICATION.1	P15.5
Digital pin 22	XIO.3	P14.0
Digital pin 23	XIO.4	P14.1
Digital pin 24	XIO.5	P15.6
Digital pin 25	XIO.6	P00.0
Digital pin 26	XIO.7	P00.8
Digital pin 27	XIO.8	P00.1
Digital pin 28	XIO.9	P00.9
Digital pin 29	XIO.10	P00.2
Digital pin 30	XIO.11	P00.10
Digital pin 31	XIO.12	P00.3
Digital pin 32	XIO.13	P00.11
Digital pin 33	XIO.14	P00.4
Digital pin 34	XIO.15	P00.12
Digital pin 35	XIO.16	P00.5
Digital pin 36	XIO.17	P33.2
Digital pin 37	XIO.18	P00.6
Digital pin 38	XIO.19	P33.1
Digital pin 39	XIO.20	P00.7
Digital pin 40	XIO.21	P33.2
Digital pin 41	XIO.22	P11.9
Digital pin 42	XIO.23	P11.10
Digital pin 43	XIO.24	P11.11
Digital pin 44 (PWM)	XIO.25	P33.3
Digital pin 45 (PWM)	XIO.26	P11.2
Digital pin 46 (PWM)	XIO.27	P11.3
Digital pin 47	XIO.28	P11.6
Digital pin 48	XIO.29	P21.3 (B-step) P21.1 (A-step) via link
Digital pin 49	XIO.30	P21.0
Digital pin 50 (MISO)	XIO.31	P33.4 + P21.3
Digital pin 51 (MOSI)	XIO.32	P2.8 + P13.3
Digital pin 52 (SCK)	XIO.33	P33.5 + P15.8
Digital pin 53 (SS)	XIO.34	P11.12 + P20.10
SPI connector 1	MISO3	P22.1
SPI connector 2	+5V	
SPI connector 3	SPCK3	P22.3
SPI connector 4	MOSI3	P22.0
SPI connector 5	RESET	
SPI connector 6	GND	
I2C SDA1	PWMH.10	P13.1
I2C SCL1	PWMH.9	P13.2
GND	PWMH.7	GND

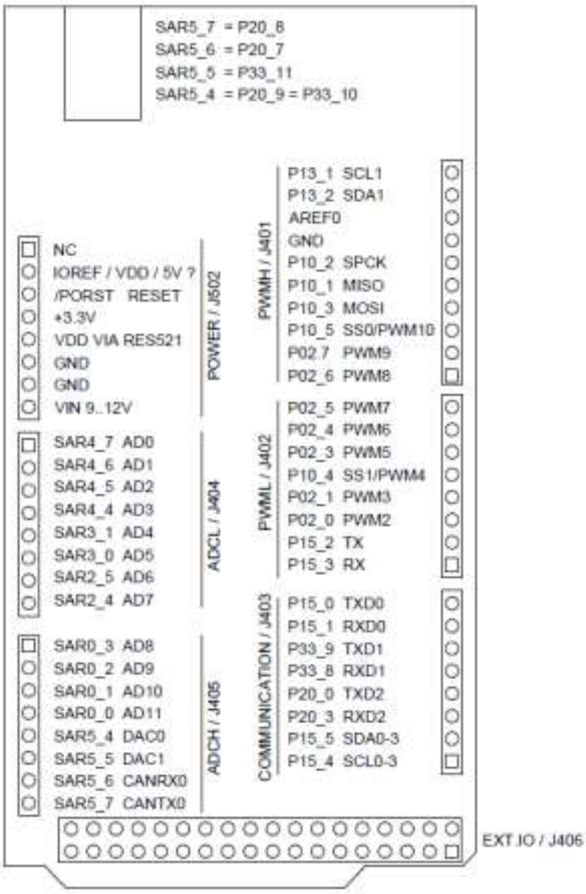


Figure 7 TC275 to Arduino Connector Mapping

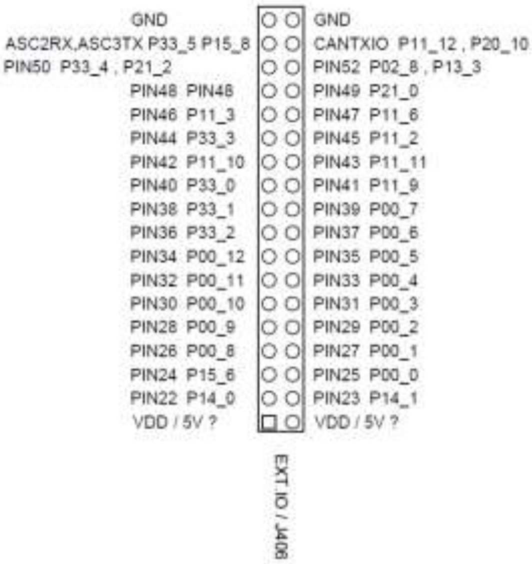


Figure 8 TC275 to Arduino EXT IO Connector Mapping

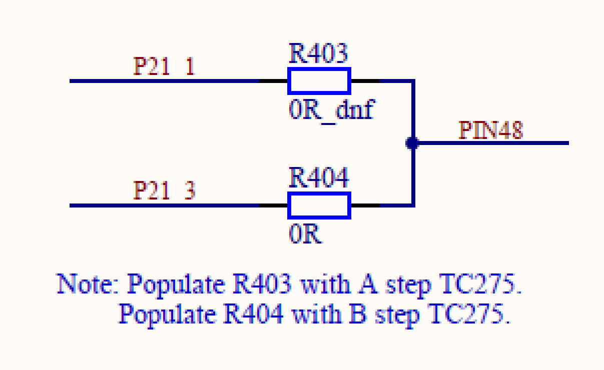
5 Powering The ShieldBuddy

The ShieldBuddy can be powered from USB or from 7-12V on the jack socket. The switch S501 allows the power source to be selected. The jumper position towards the centre of the board selects USB power.

It is possible to power the board from just the USB however some shields require more current than can be supplied via USB so in the case, the external power jack should be used.

When powered from the external jack, the CIC61508 has its own independent power regulator, as required by the safety architecture. When using USB power, the CIC61508 shares is power supply with the Aurix. This is the non-safety configuration, for development use only.

5.1 Selectable Options



5.2 Restoring an ShieldBuddy with a completely erased FLASH.

If the TC275 PFLASH becomes completely erased or if the bootmode headers are damaged, the device can no longer be accessed via JTAG or DAP. The debugger will report "No Valid ABM On Target" and the FLASH cannot be programmed, even though it might appear to have been. To overcome this, JP201 can be used to temporarily enable the debug interface so that the PFLASH can be reprogrammed. To do this, follow the procedure given below:

With the ShieldBuddy powered up:

1. Close the jumper JP201
2. Press the reset button
3. Remove the jumper on JP201
4. Attempt to start your Flash programming tool or debugger
5. The tool should now connect
6. Reprogram the PFLASH in the usual way

6 Underside Component Placement

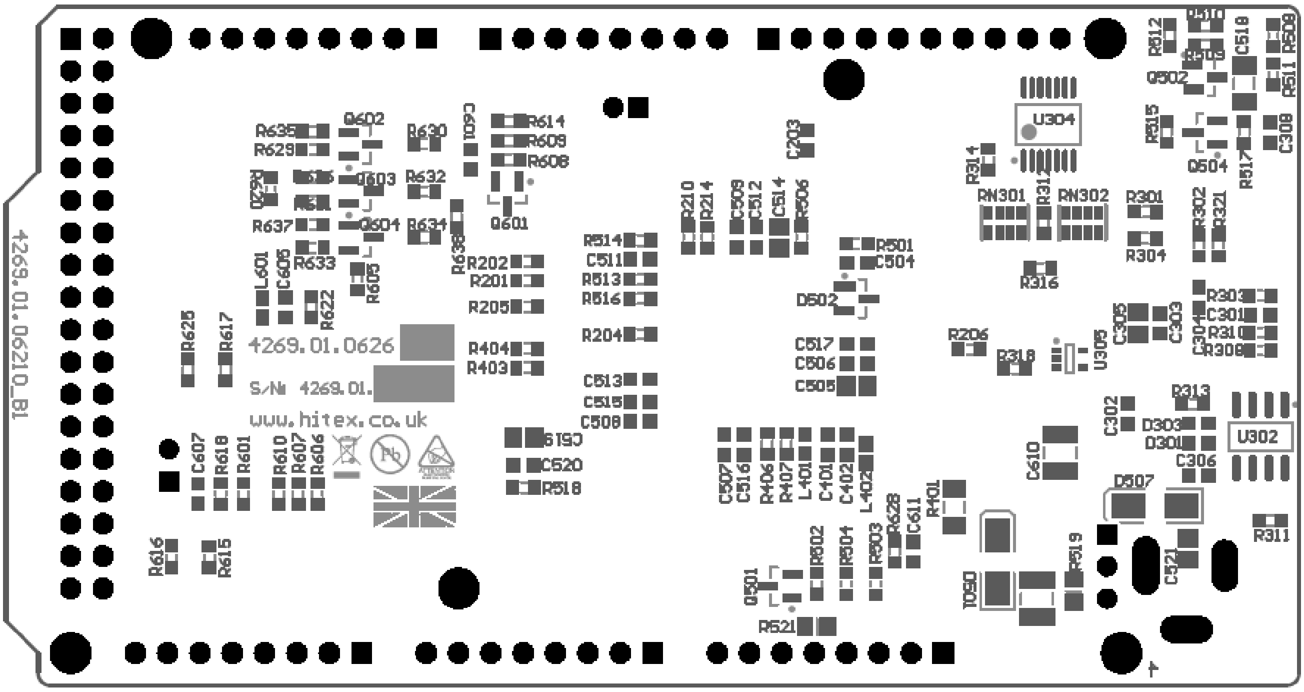


Figure 9 Bottom view of ShieldBuddy

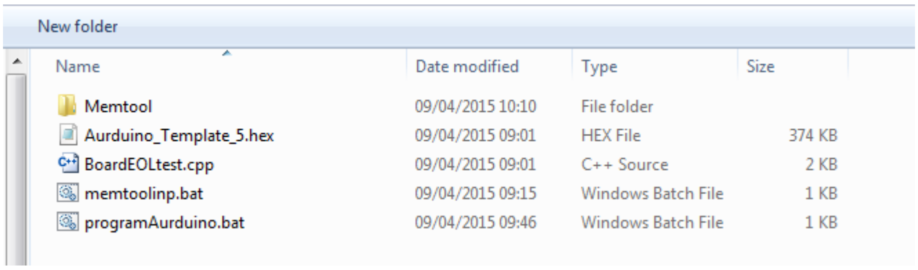
7 Appendices

7.1 Basic Board Test

If you think your ShieldBuddy has been damaged, please run this simple test to see if the CPU is still OK.

Note: It is assumed that Infineon DAS v4.6 or later is already installed on your PC.

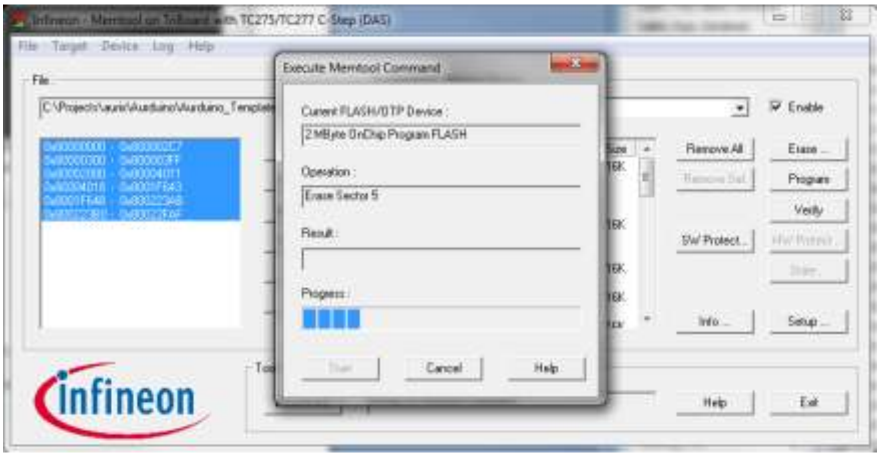
Go to “C:\Hitex\AURduinoIDE\Tools\EOL_Test”



Connect the ShieldBuddy to the USB port on your PC. Make sure that the jumper by the power jack socket is in the “USB” position.

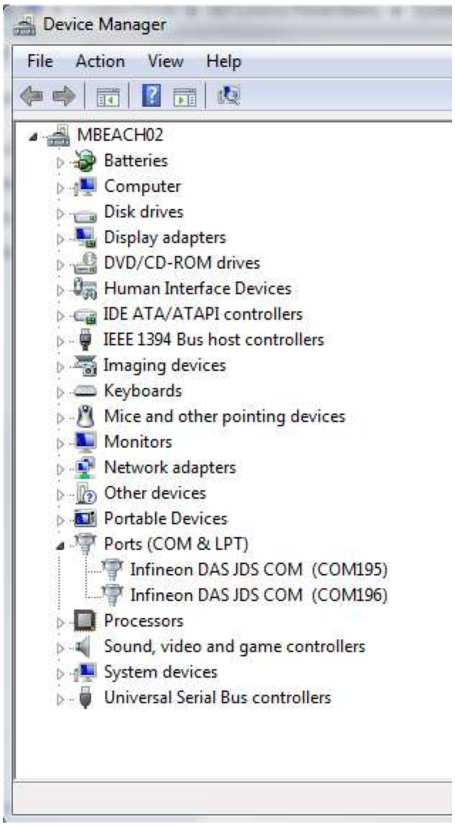
Wait for DAS to detect the ShieldBuddy – this takes about 15 seconds. Run the batch file “programShieldBuddy.bat”.

The Memtool programming tool will start and program the ShieldBuddy_Template_5.hex hexfile into the TC275 FLASH.

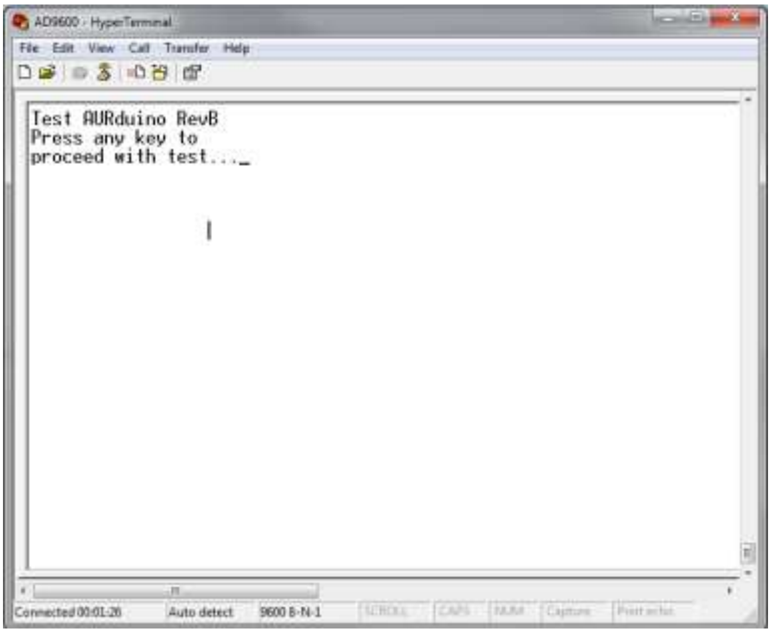


Now start a terminal emulation program (e.g. putty, MTTY, Hyperterm etc.) and use the following serial parameters:

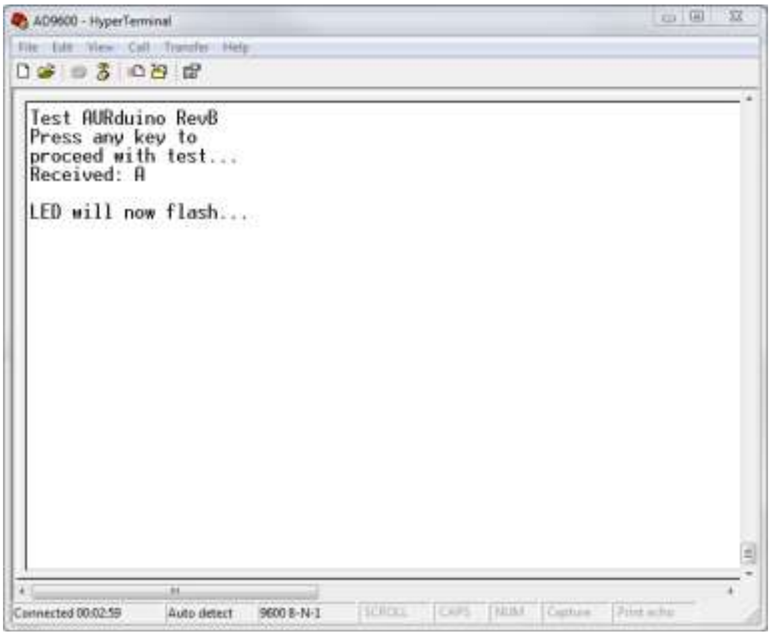
The COM port created by the ShieldBuddy will vary from PC to PC. You can find it in the Windows Control Panel - Device Manager under Ports (COM & LPT). Here it is COM196. Note it is usually the second virtual comport that is the active one. Make sure your terminal program is using this comport!



With the terminal program running, press the reset button on the ShieldBuddy. The following text should appear:



Now press any alpha key – here it was 'A'. The key you pressed will be printed into the terminal and the LED on the ShieldBuddy pinD13 should start to flash.



That completes a successful test.

www.hitex.co.uk

Published by Hitex (U.K.) Limited. 4269.40100, V1.4