ShieldBuddy TC375 User Manual

Basic information on the ShieldBuddy TC375 development board

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

Released

User Manual
4269.40100, 1.0, 2020-04

User Manual

Edition 2020-04

Published by:
Hitex (U.K.) Limited.
University Of Warwick Science Park, Coventry, CV4 7HS, UK
© 2020 Hitex (U.K.) Limited.
All Rights Reserved.

Legal Disclaimer

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics. With respect to any examples or hints given herein, any typical values stated herein and/or any information regarding the application of the product, Hitex (UK) Ltd. hereby disclaims any and all warranties and liabilities of any kind, including without limitation, warranties of non-infringement of intellectual property rights of any third party.

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**
20/4/2020	1.0	M Beach	First version, based on TC375 v3.2

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

1 Getting Started 8
1.1 What Are The ShieldBuddy TC375 Tools? 8
1.2 Getting Started With The TC375 Toolchain 8
1.3 Using The ShieldBuddy TC375 9
1.4 Using The Eclipse IDE 9
1.5 Debugging Programs Using Eclipse PLS UDE Debug Perspective 10
1.6 Getting Help 10

2 ShieldBuddy TC375 Extensions To The Arduino IDE 11
2.1 How is the ShieldBuddy Different To Other Arduinos? 11
2.2 TC375 Processor Architecture 11
2.3 Serial Ports 12
2.3.1 Compiling Existing Arduino Sketches 13
2.4 Multicore Programming Extensions 14
2.4.1 Arduino IDE Extensions 14
2.4.2 Inter-Core Communications 14
2.4.2.1 Inter-Core Communications Example 15
2.4.2.2 Using Interrupts To Coordinate and Communicate Between Cores. 17
2.4.3 Timers/Ticks/delay(), millis(), micros() Etc. 18
2.4.3.1 Core 1 19
2.4.3.2 Core 2 19
2.4.3.3 Direct Fast Access To The System Timer0 19
2.4.4 Managing the Multicore Memory Map 20
2.5 Peripheral And IO Extensions 23
2.5.1 Fast digitalRead & digitalWrite 23
2.6 Using The Analog A0 – A11 Pins As Digital IO 23
2.6.1 attachInterrupt() Function 23
2.6.2 Enabling and Disabling Interrupts 23
2.6.3 ADC Read Resolution 23
2.6.4 analogWrite() & AnalogOut 24
2.6.5 Available PWM Channels 24
2.6.5.1 PWM Frequency 24
2.6.5.2 Custom PWM Frequencies 24
2.6.5.3 Fast Update Of AnalogOut() Function 24
2.6.5.4 DAC0 and DAC1 pins 25
2.7 CAN 26
2.7.1 CAN Functions Usage 26
2.7.1.1 Receiving any message regardless of message ID 28
2.8 I2C/Wire Pins & Baudrate 28
2.9 EEPROM Support 29
2.10 Resetting The ShieldBuddy 29
2.10.1 Resetting From Programs 29
2.10.2 Resetting From Windows 29
2.11 SPI Support 30
2.11.1 Default Spi 30
2.11.2 Spi Channel 1 30
2.11.3 Spi Channel 2 30
2.11.4 Software Bit-Bashed SPI 31
2.12 Aurix DSP Function Library 32
2.13 Ethernet BootLoader/In Application Ethernet Flash Programmer 33
2.13.1 Overview 33
2.13.2 Setting The Network Addresses 33
2.13.3 Configuring The SPI 33
2.14 Using The Bootloader 34
2.15 Sending Programs To The ShieldBuddy 34
2.16 Tone() Functions 35
2.17 PWM Measurement Functions 35

2.17.1 Using The PWM Measurement Functions 35
2.18 General Purpose Timer Interrupts 37

3 Hardware Release Notes HW Revision B 38
3.1 ShieldBuddy TC375 RevA Known Problems 38
3.2 VIN Pin 38

4 Arduino-Based Connector Details 39
4.1 Board Layout 39
4.2 Connector Pin Allocation 40
4.3 TC375 ASCLIN to ShieldBuddy connector mapping 42

5 Powering The ShieldBuddy 44
5.1 Selectable Options 44
5.2 Restoring an ShieldBuddy with a completely erased FLASH. 45

6 Component Placement 46

7 Appendices 48
7.1 Basic Board Test 48

List of Figures

Figure 1 TC375 Internal Layout....11
Figure 2 TC375 RAMs....20
Figure 3 Top view of ShieldBuddy TC375....39
Figure 4 Extended IO Connector....41
Figure 5 SPI Connector....41
Figure 6 On-Board CAN Connector....41
Figure 7 Component Location – Top Side....46
Figure 8 Component Location – Underside....46

List of Tables

Table 1 SPI Names 30
Table 2 Pins available for tone() function 35
Table 3 Pins available for PWM measurement functions 35
Table 4 ASCLIN to ShieldBuddy connector mapping 42
Table 5 Arduino To ShieldBuddy To TC375 Mapping 42

1 Getting Started

1.1 What Are The ShieldBuddy TC375 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 functions e.g. digitalWrite(), analogRead(), Serial.print() etc.. These functions are implemented for all three TC375 cores and can be used without restriction.

Given the awesome power of the TC375 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 TC375 without knowing anything about the bits and bytes of the peripherals!

1.2 Getting Started With The TC375 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, 300MHz 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 7 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.

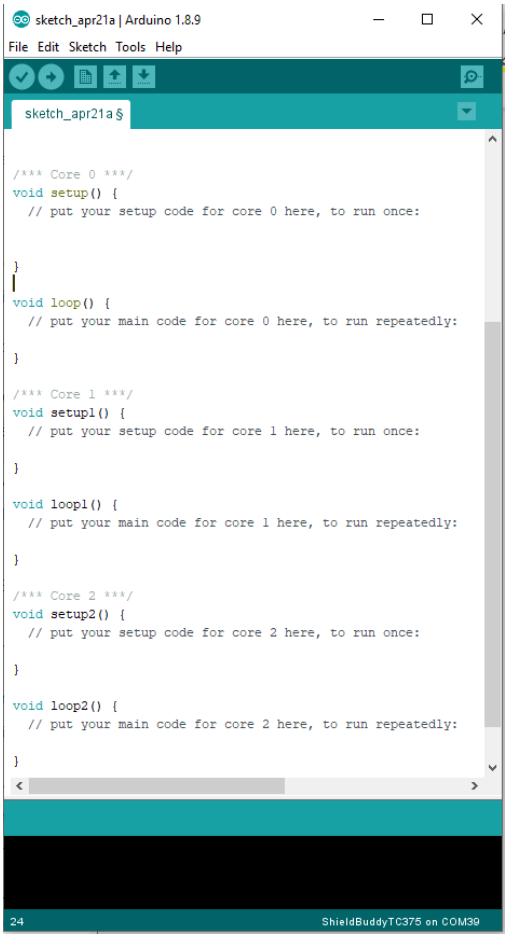
(iii) The standard Arduino IDE installed from:
https://www.arduino.cc/en/Main/Software

You may already have this but make sure it is installed in the default directory!

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

http://www.hitex.co.uk/fileadmin/uk-files/downloads/ShieldBuddy/ShieldBuddyTC375IDE.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.



Run the installer and use the password "ShieldBuddy" to copy the IDE onto your PC. Install these in the order Arduino IDE (you may already have this), Aurix freetoolchain, ShieldBuddy IDE.

1.3 Using The ShieldBuddy TC375

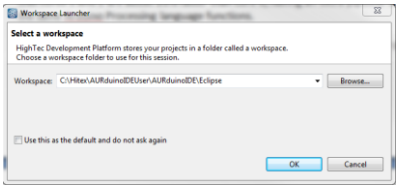
Once all of the above packages have been installed, use the ShieldBuddy just like any other Arduino except that you have three processors to play with rather than just one. Processor core 0 will run setup() and loop() with processor cores 1 and 2 running setup1()/setup2() and loop1()/loop2(). There are 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. When choosing which ShieldBuddy to use in the Arduino IDE choose "ShieldBuddy TC375".

NOTE: If you have worked with normal Arduinos, you may be used to your programs automatically running as soon as they have downloaded. In the case of the ShieldBuddy, following download, you have to press the Reset button to launch the new program. (The program will run automatically in the future when power is applied to the system.)

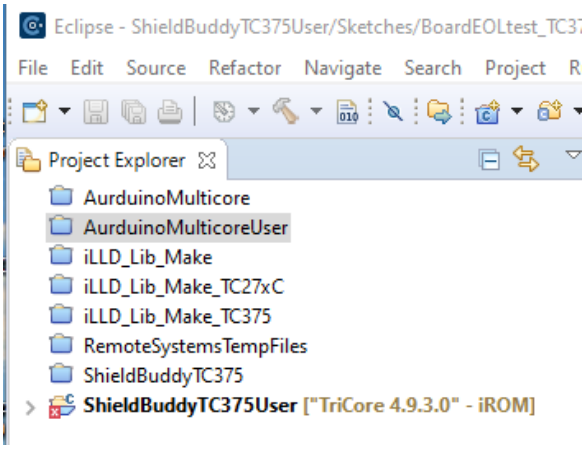
1.4 Using The Eclipse IDE

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 ShieldBuddyTC375User:



Arduino-style sketches are stored in the Sketches directory. The default sketch "BoardEOLtest_TC375.cpp" is a simple program that uses all three cores. You can overwrite the statements we used with your own. Other useful sketches are stored in the "unused" subdirectory.

To get your programs into the ShieldBuddy, use the PLS UDE debugger in its standalone mode.

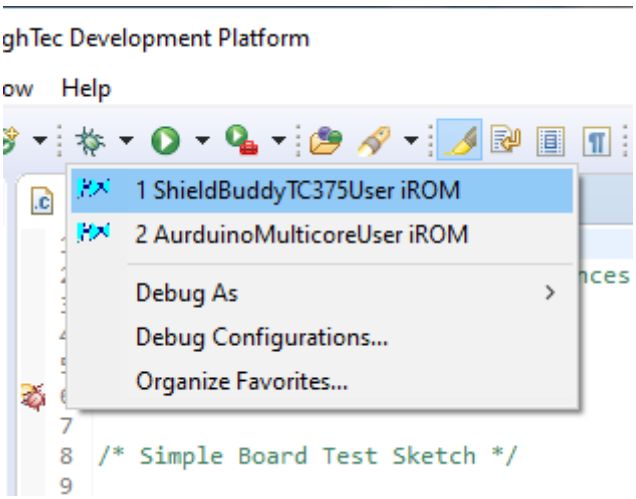
Open the workspace (ShieldBuddy with TC375 processor):

"C:\Hitex\AURduinoIDE\Eclipse\ ShieldBuddyTC375User \.ude\ShieldBuddyWorkspace_37xA.wsx"

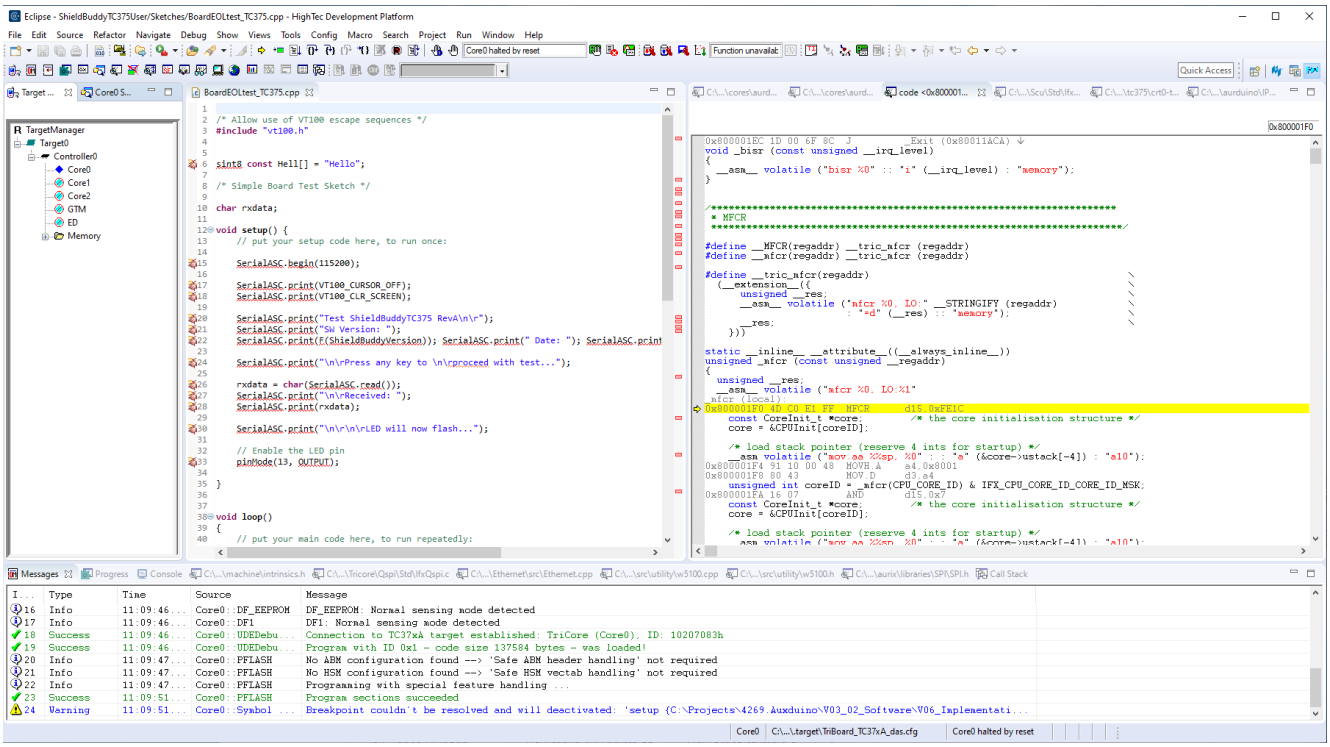
The program will automatically load. You can run it by clicking the  icon and stop it with the  icon. To reset the program, use the  icon. You can find more information on using the Eclipse tools and the PLS UDE debugger in the guide supplied with the FreeToolChain.

1.5 Debugging Programs Using Eclipse PLS UDE Debug Perspective

To debug programs in the UDE debugger, an Eclipse perspective is provided as an alternative to using it as a standalone debugger.. Debugging is started by clicking on the debug icon and then selecting "ShieldBuddy TC375 User iROM". Make sure that the ShieldBuddy is connected to the USB port on your PC first.



You can now use the UDE debug perspective in the usual; Eclipse manner.



1.6 Getting Help

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

2 ShieldBuddy TC375 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 TC375 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 car. 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 TC375 processor, there are some special macros and functions available.

2.2 TC375 Processor Architecture

Unlike the AVR, SAM3 etc. used on normal Arduinos, the TC375 has three near-identical 300MHz 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.

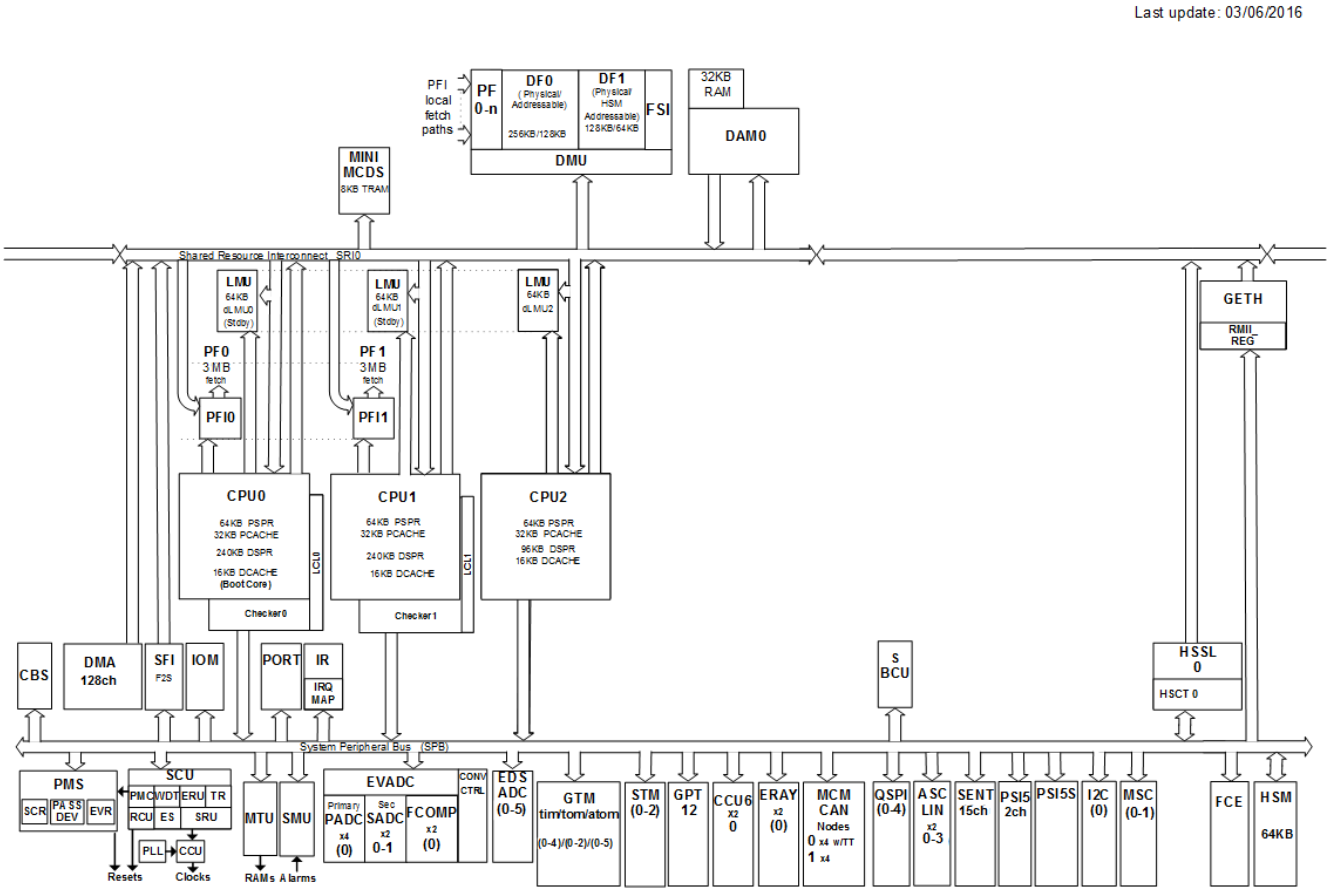


Figure 1 TC375 Internal Layout

The TC375 CPU core design has a basic 3.3ns cycle time which means you can get typically around 250 to 300 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.

With so much computing horsepower available, the TC375 can manage a huge range of peripherals. Besides commonplace peripherals like CAN, ADC, I2C, Ethernet, SPI etc. the TC375 has possibly the most powerful signal measurement and generation block to be found on any microcontroller (GTM) plus 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 ("hood" USA) 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 30 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 16 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.

The ShieldBuddy supports the following parity types:

SERIAL_8N1
SERIAL_8N2
SERIAL_8E1
SERIAL_8E2
SERIAL_8O1
SERIAL_8O2

For example:

```
SerialASC.begin(9600, SERIAL_8E1);
```

For even parity. The default is SERIAL_8N1.

2.3.1 Compiling Existing Arduino Sketches

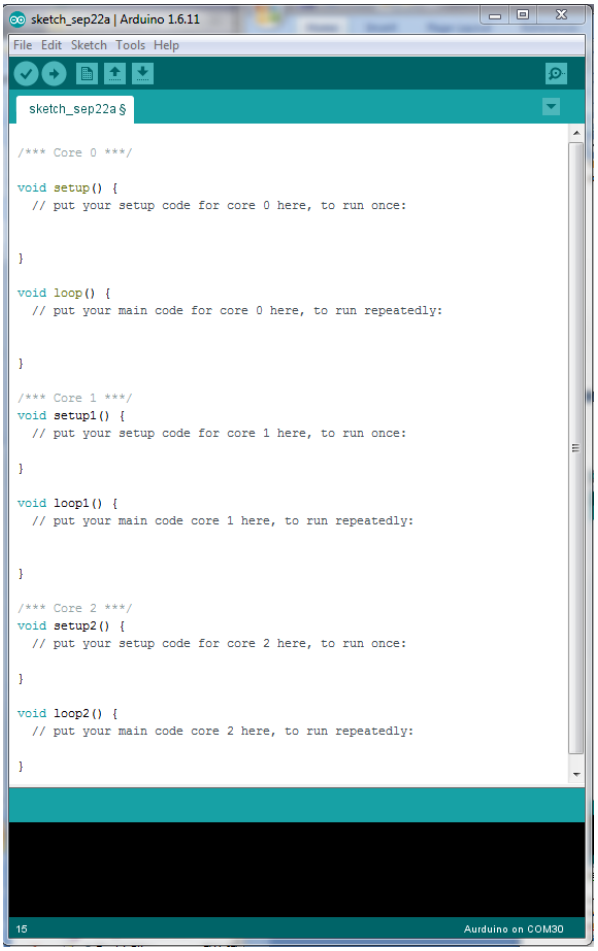
If you are compiling a sketch from a normal Arduino that uses “Serial.print()” then a quick way to make sure that the ShieldBuddy’s SerialASC is really used is to add this to the top of any source file:

```
#define Serial SerialASC  /* Redefine Arduino Serial as SerialASC */
```

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.

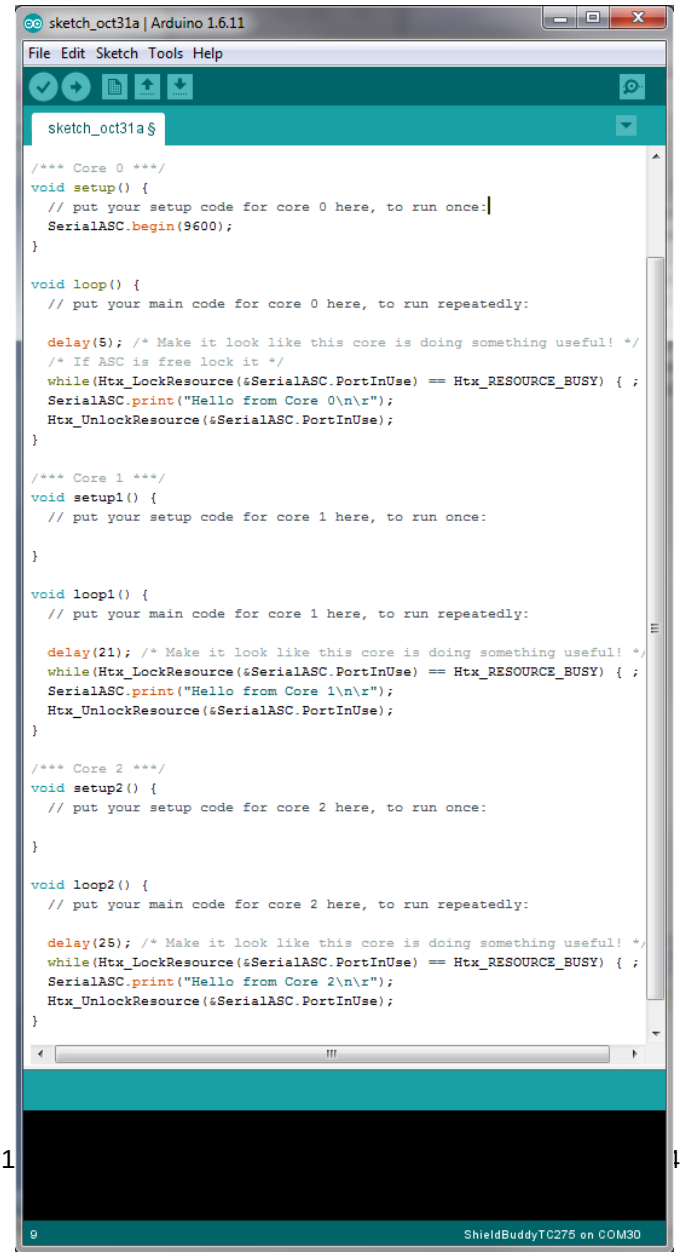
The upshot of this is that the programmer does not need to worry about cores accessing the same memory location (i.e. variable) at the same time. In some multicore processors this would cause an exception and is regarded as an error. Certainly if you are new to multicore programming, this makes life 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.



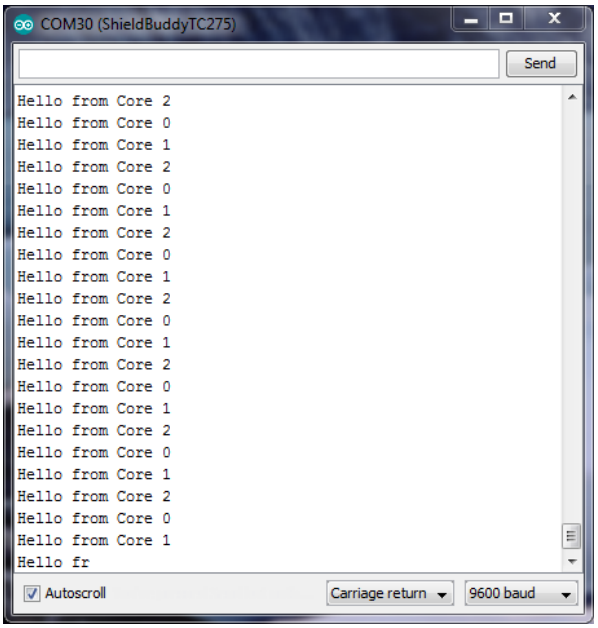
What we need to do is make sure that each core waits in turn for the other cores to finish writing to the serial port. This is quite easy using some global variables. However with true multicore programming, weird things can happen that don’t occur in single core.

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.

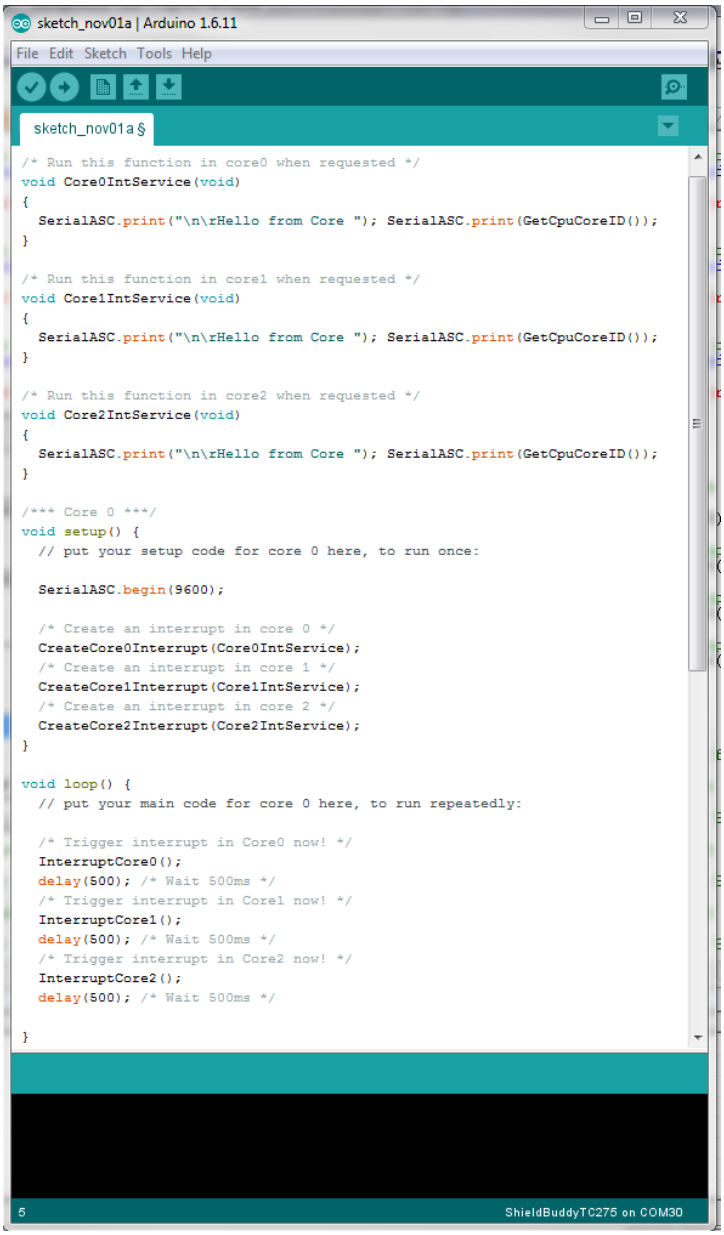
The Arduino language has been extended to allow you to trigger an interrupt in another core. This means that core 0 can trigger an interrupt in say core 1. That interrupt might tell Core 0 that a resource is now free or perhaps tell it to go and read a global variable that core0 has just updated.

```
/* 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.

Note: if you want to create periodic interrupts in one core rather than between cores, please see section 2.18.



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 TC375 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 aro und 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 TC375 can make a huge difference to the maximum performance that can be obtained.

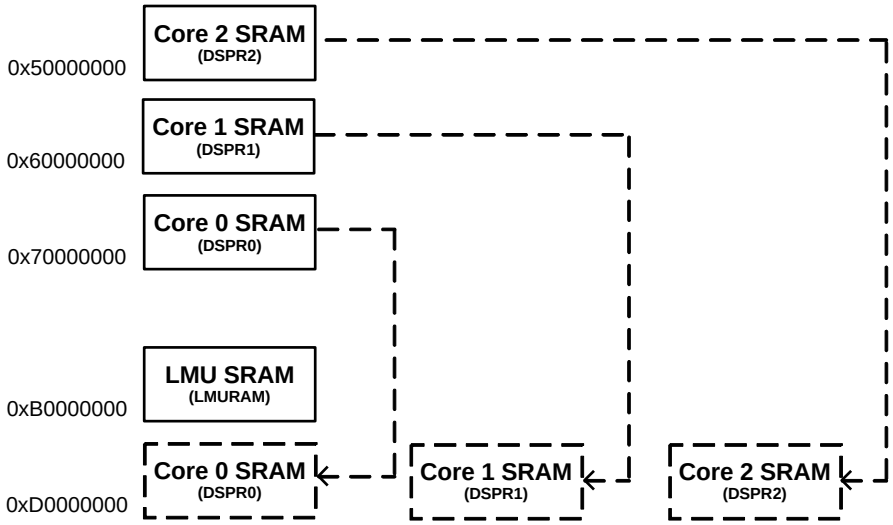


Figure 2 TC375 RAMs

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:

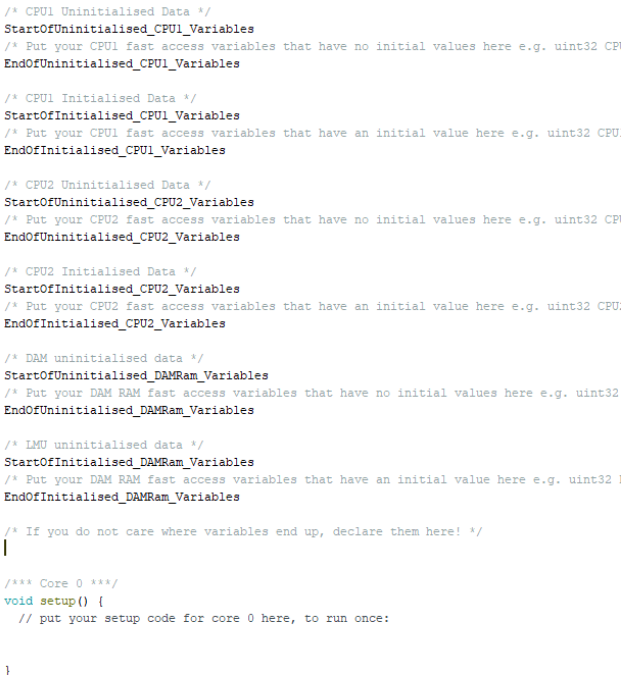

}
```

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 (DSPR0/1/2) that is visible to the other cores, albeit at reduced speed.

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

It makes sense to put the variables used by each of the cores into their local RAMs but as standard, the Ardunio IDE has no support for this. By default, any global variables will end up in the Core0 local RAM (DSPR0). 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:

```
/* DAM RAM uninitialised data */
StartOfUninitialised_DAMRam_Variables
/* Put your DAM RAM fast access variables that have no initial values here e.g. uint32 DAM_var; */
EndOfUninitialised_DAMRam_Variables

/* LMU uninitialised data */
StartOfInitialised_DAMRam_Variables
/* Put your DAM RAM fast access variables that have an initial value here e.g. uint32 DAM_var_init = 1;
*/
EndOfInitialised_DAMRam_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
```

There are some further macros that let the DMLU0/1/2 RAMs (64kbyte each) be used but these are not normally required for Arduino-style applications.

```
/* DLMU0 Core0 */
/* DLMU0 uninitialised data */
StartOfUninitialised_DLMU0Ram_Variables
EndOfUninitialised_DLMU0Ram_Variables

/* DLMU0 uninitialised data */
StartOfInitialised_DLMU0Ram_Variables
EndOfInitialised_DLMU0Ram_Variables

/* DLMU1 Core1 */
/* DLMU1 uninitialised data */
StartOfUninitialised_DLMU1Ram_Variables
EndOfUninitialised_DLMU1Ram_Variables

/* DLMU1 uninitialised data */
StartOfInitialised_DLMU1Ram_Variables
EndOfInitialised_DLMU1Ram_Variables

/* DLMU2 Core2 */
/* DLMU2 uninitialised data */
StartOfUninitialised_DLMU2Ram_Variables
EndOfUninitialised_DLMU2Ram_Variables

/* DLMU2 uninitialised data */
StartOfInitialised_DLMU2Ram_Variables
EndOfInitialised_DLMU2Ram_Variables
```

2.5 Peripheral And IO Extensions

2.5.1 Fast digitalRead & digitalWrite

These functions are identical to the Arduino versions but run much faster.

Example of writing to Pin 2.

```
digitalWrite(2,HIGH);  // 30ns
digitalWrite(2,LOW);   // 30ns => 16MHz
```

The maximum pin toggling rate is 16MHz.

2.6 Using The Analog A0 – A11 Pins As Digital IO

The Arduino pins A0 to A11 can be used as normal digital IO. They are numbered from 54 to 65 and can be used like any other IO pin. They can be accessed through these numbers or as A0 to A11 so these commands are in reality identical:

```
pinMode(54, OUTPUT);  // Set A0 to an output pin
digitalWrite(54, HIGH);   // Set A0 to '1'

pinMode(A0, OUTPUT); // Set A0 to an output pin
digitalWrite(A0, HIGH);   // Set A0 to '1'
```

2.6.1 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.

2.6.2 Enabling and Disabling Interrupts

It is possible to disable all interrupts using:

```
noInterrupts();
```

This will also stop the delay() and other timer-related functions. Interrupts can be re-enabled using:

```
interrupts();
```

2.6.3 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);
```

Typically around 1500ksamples/sec is possible, regardless of the resolution.

2.6.4 analogWrite() & AnalogOut

2.6.5 Available PWM Channels

The following pins can be used with analogWrite() to generate PWM:

2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 22, 23, 24, 25, 26, 27, 28, DAC0, DAC1

2.6.5.1 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.

2.6.5.2 Custom PWM Frequencies

It is also possible to set any PWM frequency using the useCustomPwmFreq() function:

```
/* Use 4000Hz carrier */
useCustomPwmFreq(4000);
```

The maximum frequency that may be set is 390kHz. The minimum is 6Hz.

If you want to change the PWM frequency after calling analogWrite(x,y), use the following functions:

```
AnalogOut_2_Reset();  // Allow analog channel 2 to be altered
useCustomPwmFreq(3900);  // Change to 3900Hz carrier

analogWrite(2, 128);  // Write 50% duty ratio at 3900Hz carrier
```

2.6.5.3 Fast Update Of AnalogOut() Function

In situations where the duty ratio has to be updated very frequently, it is often better to update just the duty ratio register in the PWM system for the particular channel in use rather than using the normal analogWrite(). This can be done using macros like:

```
AnalogOut_2_DutyRatio = 128;
```

The value used must be within the range allowed by the resolution you are using. For the default 8-bit, this is 0-255; for 10-bit this is 0-1023 and so on.

For this to work, you must have used the normal analogWrite(x, y) for that channel at least once e,g.

```
analogWrite(2, 128);
```

2.6.5.4 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
```

2.7 CAN

Controller Area Network is available on the three pin headers P202. This has a CAN transceiver and so CANH and CANL are available, with a 120R terminating resistor fitted. This is CAN channel whose software name is "CAN".

Controller Area Network is supported via the CANRX/CANTX pins, J406 (double row connector) pins 23 and 22 plus J406 pin53 and J405 DAC0. These are CAN0, CAN1 and CAN3 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 TC375's real capability and is limited for the sake of simplicity.

There are three CAN channels on the ShieldBuddy TC375, CAN0, CAN1 and CAN3. These are located as follows:

Name	TC375 Port	ShieldBuddy Pin	Software Name
CANH/CANL	P23.0/P13.0	P202 pin2 & pin1	CAN
CAN0 RX	P20.7	pin CANRX	CAN0
CAN0 TX	P20.8	pin CANTX	
CAN1 RX	P14.1	J406 pin23	CAN1
CAN1 TX	P14.0	J406 pin22	
CAN3 RX	P20.9	J405 DAC0	CAN3
CAN3 TX	P20.10	J406 pin53	

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

2.7.1 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.

For the send message function, we must provide a structure into which the data to be sent can be placed. The predefined structure type "CANMessagePayloadType" may be used for this.

```
typedef union { uint8 bytes[8]; uint32 dword[2]; } CANMessagePayloadType;
```

The union "msg1" is created:

```
CANMessagePayloadType msg1;
```

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 */
msg1.dword[0] = 0x12340000;
msg1.dword[1] = 0x9abc000;

CAN0_SendMessage(0x100, &msg1, 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 "CANMessagePayloadType" can be used for this:

```
CANMessagePayloadType msg1;
```

The data received can be accessed in:

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

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

```
IfxCan_Status RxStatus;
```

The return values are any one of:

```
    IfxCan_Status_ok                = 0x00000000,
    IfxCan_Status_notInitialised    = 0x00000001,
    IfxCan_Status_busOff            = 0x00000010,
    IfxCan_Status_notSentBusy       = 0x00000020,
    IfxCan_Status_receiveEmpty      = 0x00000040,
    IfxCan_Status_messageLost       = 0x00000080,
    IfxCan_Status_newData           = 0x00000100,
    IfxCan_Status_newDataButOneLost = 0x00000180
```

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. When this is data, they will return a value of 0x100.

2.7.1.1 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);
```

2.8 I2C/Wire Pins & Baudrate

The ShieldBuddy's default I2Cperipheral 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. A further set are on pins SDA1 and SCL1.

```
Wire.setWirePins(UsePins_20_21); // Default pins for Arduino Due/MEGA SCL1/SDA1
```

Or:

```
Wire.setWirePins(UsePins_6_7); // Pins 6 & 7
```

Or:

```
Wire.setWirePins(UsePins_SDA1_SCL1); // SDA1, SCL1

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)
```

Only one set of pins can be used with the Wire library at once. If you need two I2C channels then the second one will have to use the software-driven I2C library. To do this, the SoftwareWire.h must be included at the top of the file, for example:

```
// Use SW I2C port on any two pins
#include <SoftwareWire.h>

// Create Software I2C on pin6 (SDA) pin 7 (SCL)
SoftwareWire SwWire( 6, 7, 0, 0); /* No pullups, no clock stretch */
```

It can then be used just like the normal I2C ports except that the Baudrate is fixed at around 100kbit/s.

2.9 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 TC375 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.

2.10 Resetting The ShieldBuddy

2.10.1 Resetting From Programs

It is possible to reset the ShieldBuddy by executing the Reset_TC375() function. This causes a TC375 system reset which puts the CPUs into the reset state.

2.10.2 Resetting From Windows

To reset and run the program in the ShieldBuddy, there is a simple DOS command-line utility Reset_TC375.exe that will allow the board to be manually reset and then run. This is located in:

```
C:\Program Files (x86)\Arduino\hardware\ShieldBuddyTC375\aurix\variants\tc375
```

It is a good idea to create shortcut to the Reset_TC375.exe on the Windows Taskbar so that you can manually reset and run the ShieldBuddy after downloading a program.

2.11 SPI Support

2.11.1 Default Spi

The SPI is similar to that on the Arduino Uno and MEGA and can be used in much the same way. The default slave select is pin10 with alternative one being on pin4. These can be used in the same way as on the Arduino.

```
Spi.begin();   // Use default slave select on p10
Spi.begin(10); // Use default slave select on p10
Spi.begin(4);  // Use slave select on p4 (used by SD cards)
Spi.begin(5);  // Use slave select on p5 (used by TFT shields)
```

2.11.2 Spi Channel 1

There is a second independent Spi channel on p12 (MISO), p11(MOSI) and p13 (SCK) which also uses p10 as the slave select. To use this Spi channel:

```
Spi.begin(BOARD_SPI_SS0_S1);   // Use Spi1 with slave select on p10
```

To use this SPI channel:

```
Spi.transfer(BOARD_SPI_SS0_S1, data);
```

2.11.3 Spi Channel 2

There is a further Spi channel on p50 (MISO), p51 (MOSI) and p52 (SCK) which currently implemented as a bit-bashed Spi. This is intended for use with special shields like the Industrial Shield range from Boot & Work. Two possible slave selects are supported, pin53 and pin10.

```
Spi.begin(BOARD_SOFT_SPI_SS2); // Use slave select on p53
Spi.begin(BOARD_SOFT_SPI_SS0); // Use slave select on p10
```

To use these channels:

```
Spi.transfer(BOARD_SOFT_SPI_SS2), data);
Spi.transfer(BOARD_SOFT_SPI_SS0), data);
```

Note that the latter cannot be used at the same time as any other Spi channel that has p10 as the slave select. This Spi channel runs at about 3Mbit/s so a typical 8-bit transfer takes around 2.9us.

Table 1 SPI Names

SPI Name	Comment	Used Pins
BOARD_SPI_SS0	Pin10 is default CS on SPI Ch0	MISO = P201.1, MOSI = P201.4 SCK = P201.3
BOARD_SPI_SS0_S1	Really pin 10 but this means use it with SPI Ch 1	MISO = p12, MOSI = p11 SCK = p13
BOARD_SOFT_SPI_SS0	Bit bashed SPI only p10 chip select	MISO = p50, MOSI = p51 SCK = p52
BOARD_SOFT_SPI_SS2	Bit bashed SPI only p53 chip select	MISO = p50, MOSI = p51 SCK = p52
BOARD_SPI_SS1	Used for SD Cards on SPI Ch0	MISO = P201.1, MOSI = P201.4 SCK = P201.3

2.11.4 Software Bit-Bashed SPI

A general purpose software-drive SPI is available on any 4 pins. This can be used on pins where there is no hardware SPI available.

```
#include "SPI.h"

/* Initialise 4 pins for software SPI channel */
/* SoftSPi_Begin(SS_pin, SCK_pin, MISO_pin, MOSI_pin) */
SPI.SoftSPi_Begin(2,3,4,5); // Use pins 2,3,4,5 for SPI
```

Data can be sent and received using:

```
// Send data via soft SPI
SpiData = SPI.SoftSPi_Transfer(2, SpiData, SPI_LAST);
```

The software SPI only supports SPI MODE0 (clock idle low, shift transmit data on trailing edge) with the MSB sent first. You can have as many SPI channels as there are pins available!

With complex shields like the Ethernet/SD card shield, you only need to make sure that the SPI.SoftSPi_Begin() is called at the start of setup() with the pins you want to use. This will defined a chip selects to be used for the software SPI channel. Any further chip selects you need for further devices can be simply specified in the SPI.transfer() call.

2.12 Aurix DSP Function Library

The Aurix has a number of built-in DSP-like functions such as saturated maths, Q-arithmetic, circular buffer types etc. These are often used in applications such as:

- Complex Arithmetic
- Vector Arithmetic
- FIR Filters
- IIR Filters
- Adaptive Filters
- Fast Fourier Transforms
- Discrete Cosine Transform
- Mathematical functions
- Matrix operations
- Statistical functions

To allow these to be implemented easily and efficiently, Infineon have released the “TriLib” library. This consists of assembler-coded routines that are highly optimized for minimum run time and are designed to be callable directly from C and C++ programs (including the Arduino IDE). They are not floating point. For such operations, the on-board floating point units are directly used by the compiler, so nothing special needs to be done. It should be borne in mind though that the free Aurix GCC toolchain used with the ShieldBuddy does not have the highly optimised runtime libraries supplied with the full version so some functions are slower than might be expected.

It is recommend to use the DSP TriLib functions on cores 1 or 2 as these are around 20% faster than core0 due to the more sophisticated pipeline. There are no special steps to take when using them.

2.13 Ethernet BootLoader/In Application Ethernet Flash Programmer

It is possible to program the TC375 PFLASH via an Ethernet shield using TFTP. There are a number of ways to do this which are described below.

2.13.1 Overview

TFTP is a simple file transfer protocol which is often used to boot diskless and embedded systems. It uses UDP and requires a conventional TCP/IP network setup. The booloader is not 100% robust and if transmission and programming is interrupted, it is possible for the ShieldBuddy to be left with an incomplete or damaged application in its Pflash. If this happens, you will have to reprogram it directly from the Arduino IDE or the UDE debugger via USB.

2.13.2 Setting The Network Addresses

The default IP address "192.168.3.177" will need to be changed to suit your local network environment, as will the IP addresses given in .\aurix\system\include\net.h. Edit this file as required.

```
/* Gateway Address */
#define GWIP0     192
#define GWIP1     168
#define GWIP2     3
#define GWIP3     1

/* Subnet Mask */
#define MASK0     255
#define MASK1     255
#define MASK2     255
#define MASK3     0

/* MAC Address */
#define MAC0      0x12
#define MAC1      0x34
#define MAC2      0x45
#define MAC3      0x78
#define MAC4      0x9A
#define MAC5      0xBC

/* IP Address */
#define IP0       192
#define IP1       168
#define IP2       3
#define IP3       177
```

2.13.3 Configuring The SPI

The Ethernet bootloader use a bit-based SPI to eliminate the need for interrupts running during PFlash programming. This can be configured to use any 4 Arduino pins but the most commonly used ones for Ethernet are given in net.h. Please note that the SPI pins on the 6-way ICP (P201) connector are not normally assigned Arduino pin numbers but on the ShieldBuddy, they are pins 61, 62 and 63.

```
/*** Bit Bashed SPI for TFTP Download ***/

/* Pin definitions for ICP SPI port */
#define SCK_pin   62
#define MISO_pin  61
```

```
#define MOSI_pin  63
#define TFTP_SS   10
```

These can be changed as required.

2.14 Using The Bootloader

The EnterBootLoader() function will wait for 10 seconds for TFTP transmission from the PC to begin. If nothing is received it times out and exits. If data is received, it is programmed into the TC375 Pflash. If the data stream is interrupted before completion, it waits for transmission to restart without timing out. This is because the Pflash is likely to be in a corrupted state and the TC375 continues to execute the Bootloader which is running from the PSPR0 RAM. If the TC375 is reset in this state, it will not restart and you will have to reprogram it via the Arduino IDE or the UDE debugger.

The Bootloader can be used in two ways:

(i) Program the "BootLoaderTest" sketch into the ShieldBuddy and then at some later time, send it the real application via Ethernet and TFTP.

(ii) Build sketches/applications with a call to the Bootloader included in them:

```
// Call the bootloader to program itself back into Flash
// Ends with a TC375 reset if programming was successful otherwise it returns
// after 10 seconds.
EnterBootLoader();
```

2.15 Sending Programs To The ShieldBuddy

The Bootloader expects to receive a binary image of the new ShieldBuddy program. At the moment this not possible from directly from the Arduino IDE. Note that CPU1 & 2 cannot be used during the programming process and the bootloader will disable them until the final reset occurs.

First of all, use the Arduino IDE to program the BootLoaderTest sketch into the ShieldBuddy in the normal way and then reset the board. The binary image of this sketch can be created with the HEX2BIN tool from the same AurduinoUpload.hex file that the Arduino IDE would normally send to the ShieldBuddy via USB:

```
C:\Hitex\AURduinoIDE\Tools\hex2bin c:\HIGHTEC\AurduinoUpload.hex
```

The binary file is then sent to the ShieldBuddy using TFTP.EXE:

```
C:\Hitex\AURduinoIDE\Tools\tftp -i 192.168.3.177 put C:\HIGHTEC\AurduinoUpload.bin
```

These lines can be combined into a batchfile containing:

```
C:\Hitex\AURduinoIDE\Tools\hex2bin C:\HIGHTEC\AurduinoUpload.hex
C:\Hitex\AURduinoIDE\Tools\tftp -i 192.168.3.177 put C:\HIGHTEC\AurduinoUpload.bin
```

Now run the batchfile:

```
C:\Hitex\AURduinoIDE\Tools\SendAppTftp.bat
```

The user LED (pin13) on the board will flash until the TFTP connection is made at which point it stops. When the programming is completed, the TC375 will be reset and the bootloader will restart. You should see the LED flashing again.

2.16 Tone() Functions

The standard Tone functions are implemented, as per the Arduino. The only difference is that the range of tones is from 0.232Hz to 100MHz. The duration can be up to 65.5 seconds.

Not all ShieldBuddy pins can be used for the Tone functions. The following pins may be used:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
18	19	20	21	22	28	30	31
32	34	39	41	42	43	47	49
51	52						

Table 2 Pins available for tone() function

2.17 PWM Measurement Functions

The TC375 GTM TIM modules can be used to make PWM period and duty cycle measurements automatically and without interrupts. The PWM frequency must be in the range of 5.96Hz to 10MHz. The following ShieldBuddy pins can be used for this purpose:

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	17	18	19	20
21	24	25	27	28	29	31	33	35	37
38	39	44	45	46					

Table 3 Pins available for PWM measurement functions

2.17.1 Using The PWM Measurement Functions

The PWM measurement system must first be initialized for the pin you want to use, here on pin 8. This is for a normal positive-going PWM signal:

```
// Initialise PWM measurement
Init_TIM_TPWM(8, TIM_TPWM_RISINGEDGE);    // Use pin 8, rising edges
```

For an inverted PWM signal on pin9 for example:

```
Init_TIM_TPWM(9, TIM_TPWM_FALLINGEDGE);  // Use pin 9, falling edges
```

The function used to make the measurement is “MeasurePwm()”. This has parameters as per:

```
MeasurePwm(uint8 _pin, uint32 *Period, uint32 *Duration, float *DutyRatio);
```

It expects to receive the address of the variable into which you want the new data to be inserted, e.g.

```
MeasurePwm(8, &PWM_Period0, &PWM_Duration0, &DutyRatio0);
```

Where the parameters have previously been declared as:

```
uint32 PWM_Period0;
uint32 PWM_Duration0;
float DutyRatio0;
```

The PWM period and duration are returned as integers scaled in units of 10ns. Thus a period of 1ms will result in PWM_Period0 being 100000. The duty ratio is returned as floating point value in the range of 0-1. Please

note that if there is just a '1' or '0' applied to the pin i.e. no PWM is present, the MeasurePwm(() function will not update the parameters passed to it. To allow you to check for this condition, it returns either "NoPwmMeasurementData" or "PwmMeasurementData" from the typdef MeasurePwmReturnType:

```
typedef enum { NoPwmMeasurementData, PwmMeasurementData } MeasurePwmReturnType;
```

For example:

```
if(MeasurePwm(8, &PWM_Period0, &PWM_Duration0, &DutyRatio0) == PwmMeasurementData)
{
   // New PWM data available – parameters passed will be updated
}
else
{
   // New PWM data available – parameters passed are not updated
}
```

To measure just the duty ratio of a PWM signal, you can use:

```
DutyRatio0 = MeasureDutyRatio(8);
```

To measure just the frequency of a PWM signal (or in fact any signal), you can use:

```
Frequency0 = MeasureFrequency(8);
```

.

2.18 General Purpose Timer Interrupts

There are nine general purpose timers available (based on the ATOMs in the GTM) that can be used to call an user-defined function from a periodic interrupts. The longest time is about 170 seconds and the shortest is around 1us. The units of time are by default based on 0.02us per count (50MHz).

Example

We want call this function every 100us using timer 2:

```
void UserTimer2Handler(int i)
{
   digitalWrite(13,!digitalRead(13));
}
```

First, set the name of the function to be called:

```
// Set user handlers
TimerChannelConfig[2].user_inthandler = UserTimer2Handler;
```

Then initialize the Timer 2 channel:

```
// Initialise general timer channels
InitialiseTimerChannel(2);
```

Finally, set the period of the interrupt, in units of 0.02us:

```
// Set period of timers
SetTimerChannelPeriod(2, 5000);
```

The function UserTimer2Handler() will now be called every 100us.

You can temporarily disable a timer channel using:

```
DisableTimerChannelInt(2);
```

And restart it with:

```
EnableTimerChannelInt(2);
```

3 Hardware Release Notes HW Revision B

3.1 ShieldBuddy TC375 RevA Known Problems

The ShieldBuddy TC375 Revision A 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 the TLF35584 device to a 3V3 version and the changing of some resistors

3.2 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 TC375 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

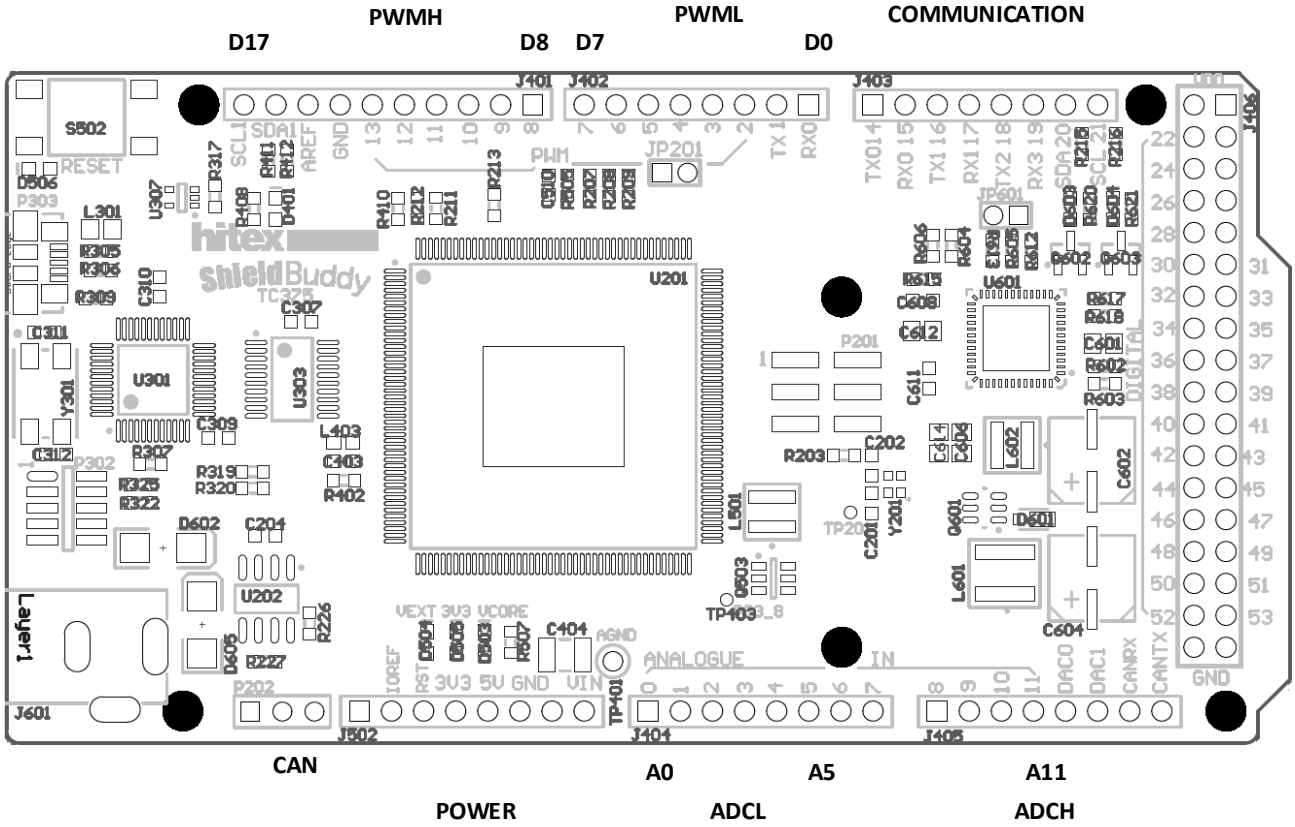
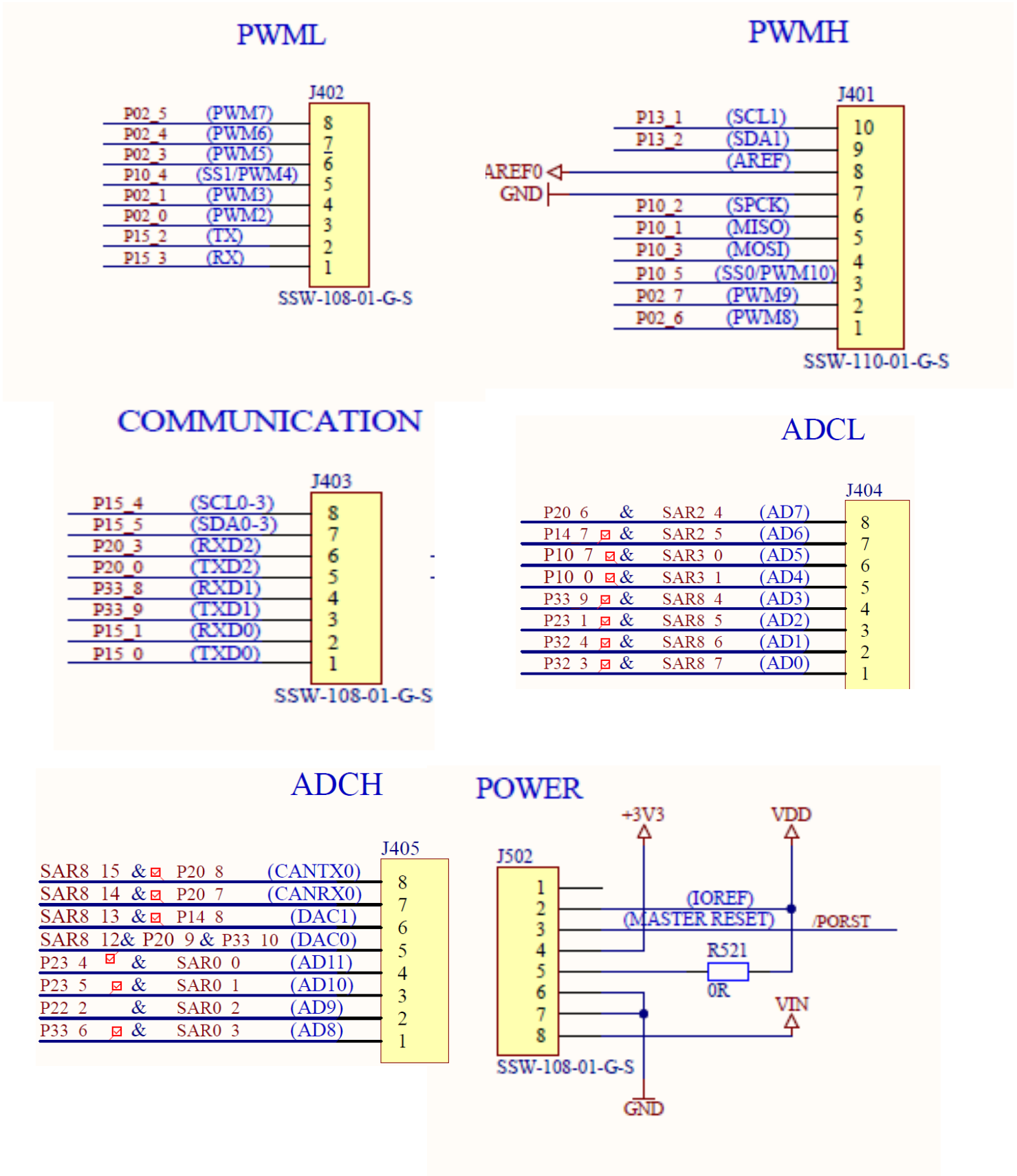


Figure 3 Top view of ShieldBuddy TC375

4.2 Connector Pin Allocation



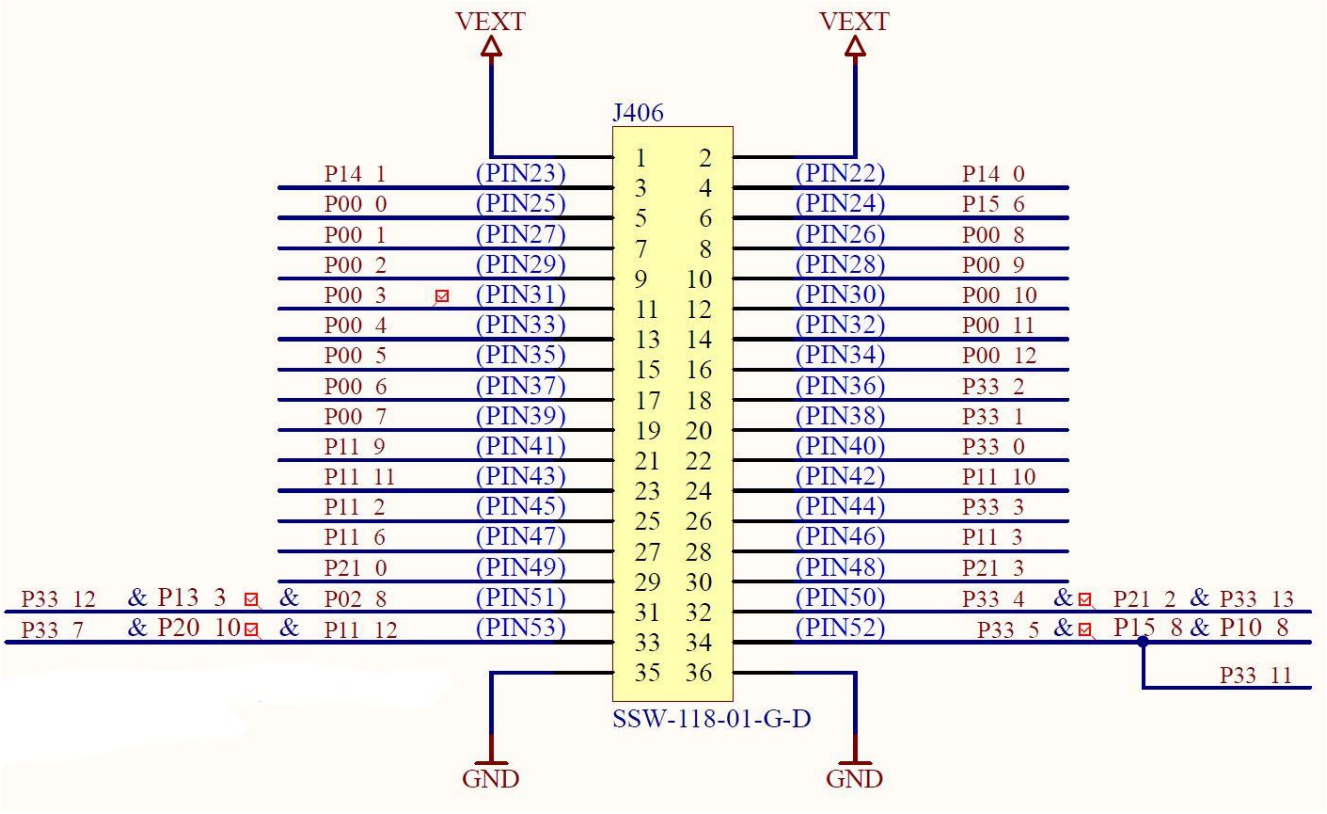


Figure 4 Extended IO Connector

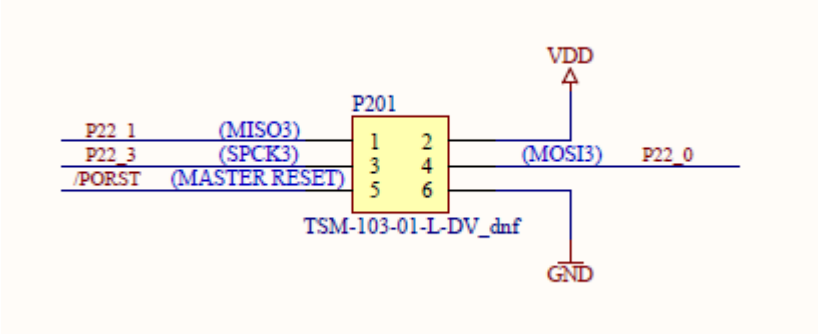


Figure 5 SPI Connector

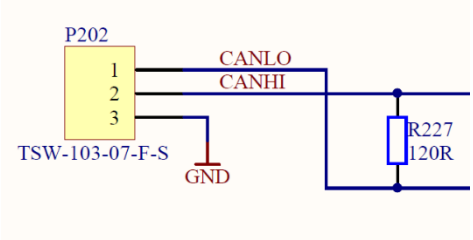


Figure 6 On-Board CAN Connector

4.3 TC375 ASCLIN to ShieldBuddy connector mapping

Table 4 ASCLIN to ShieldBuddy connector mapping

TC375 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 5 Arduino To ShieldBuddy To TC375 Mapping

Arduino Signal Name	ShieldBuddy Connector Name	TC375T Pin Assignment
Analog pin 0	ADCL.1	SAR8.7/P32.3
Analog pin 1	ADCL.2	SAR8.6/P32.4
Analog pin 2	ADCL.3	SAR8.5/P23.1
Analog pin 3	ADCL.4	SAR8.4/P33.9
Analog pin 4	ADCL.5	SAR3.1/P10.0
Analog pin 5	ADCL.6	SAR3.0/P10.7
Analog pin 6	ADCL.7	SAR2.5/P14.7
Analog pin 7	ADCL.8	SAR2.4/P20.6
Analog pin 8	ADCH.1	SAR0.3/P33.6
Analog pin 9	ADCH.2	SAR0.2/P22.2
Analog pin 10	ADCH.3	SAR0.1/P23.5
Analog pin 11	ADCH.4	SAR0.0/P23.4
Analog pin 12/DAC0	ADCH.5	SAR8.12/P20.9/P33.10
Analog pin 13/DAC1	ADCH.6	SAR8.13/P14.8
Analog pin 14/CAN RX	ADCH.7	SAR8.14/P20.7 CAN0 RX
Analog pin 15/CAN TX	ADCH.8	SAR8.15/P20.8 CAN0 TX
Digital pin 4 (PWM/SS)	PWML.5	P10.4 (P2.2 optional via R410 dnf)
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	TC375T 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
Digital pin 49	XIO.30	P21.0
Digital pin 50 (MISO)	XIO.31	P33.4 + P21.2 + P33.13
Digital pin 51 (MOSI)	XIO.32	P33.12 + P13.3 + P2.8
Digital pin 52 (SCK)	XIO.33	P33.5 + P15.8 + P10.8 + P33.11
Digital pin 53 (SS)	XIO.34	P33.7 + P20.10 + P11.12
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

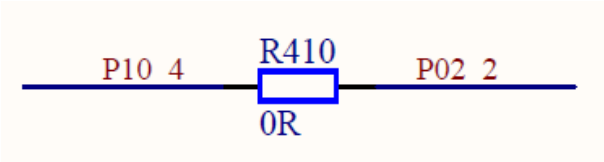
5 Powering The ShieldBuddy

The ShieldBuddy can be powered from USB or from 6V to12V on the jack socket.

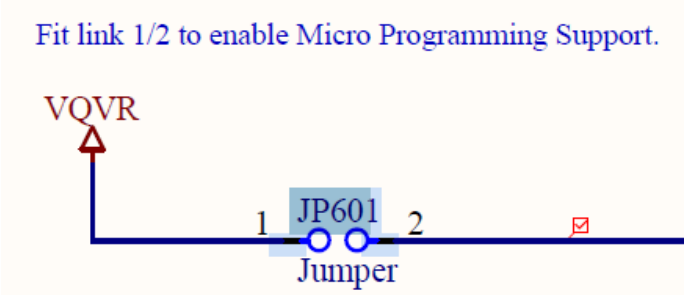
It is possible to power the board from just the USB however some shields require more current than can be supplied via USB so in the case, the external power jack should be used.

5.1 Selectable Options

It is sometimes useful to be able to access all of Port2 on consecutive pins on PWML (J402). This can be done by fitting a zero Ohm link in the R410 position.



The TLF35584 is a sophisticated power regulator and watchdog device. Unless it is refreshed or disabled within 600ms of power-on, it will cause the TC375 to reset. To prevent this, jumper JP601 is fitted. The ShieldBuddy startup code in fact does contain a SPI-based driver that disables the TLF35584 so this jumper is not essential but if you use other programming environments, this may not be present so JP601 should be fitted.



5.2 Restoring an ShieldBuddy with a completely erased FLASH.

If the TC375 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 Component Placement

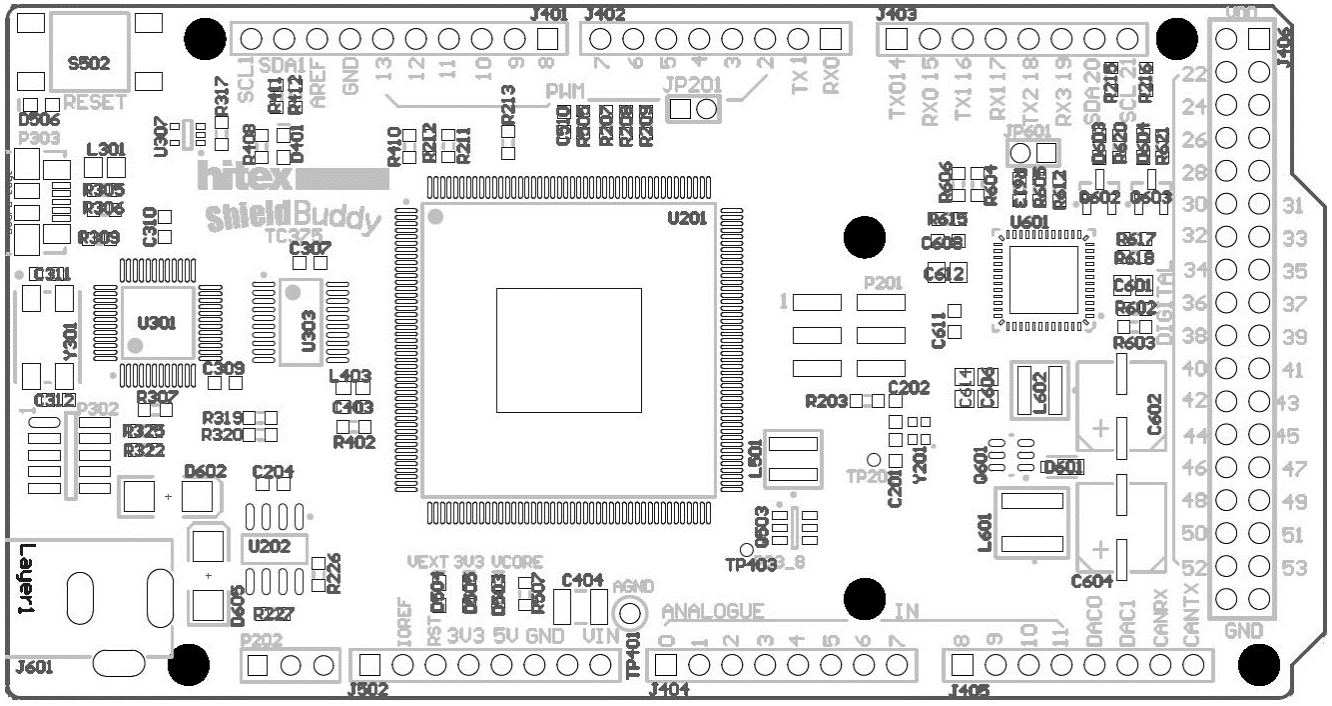


Figure 7 Component Location – Top Side

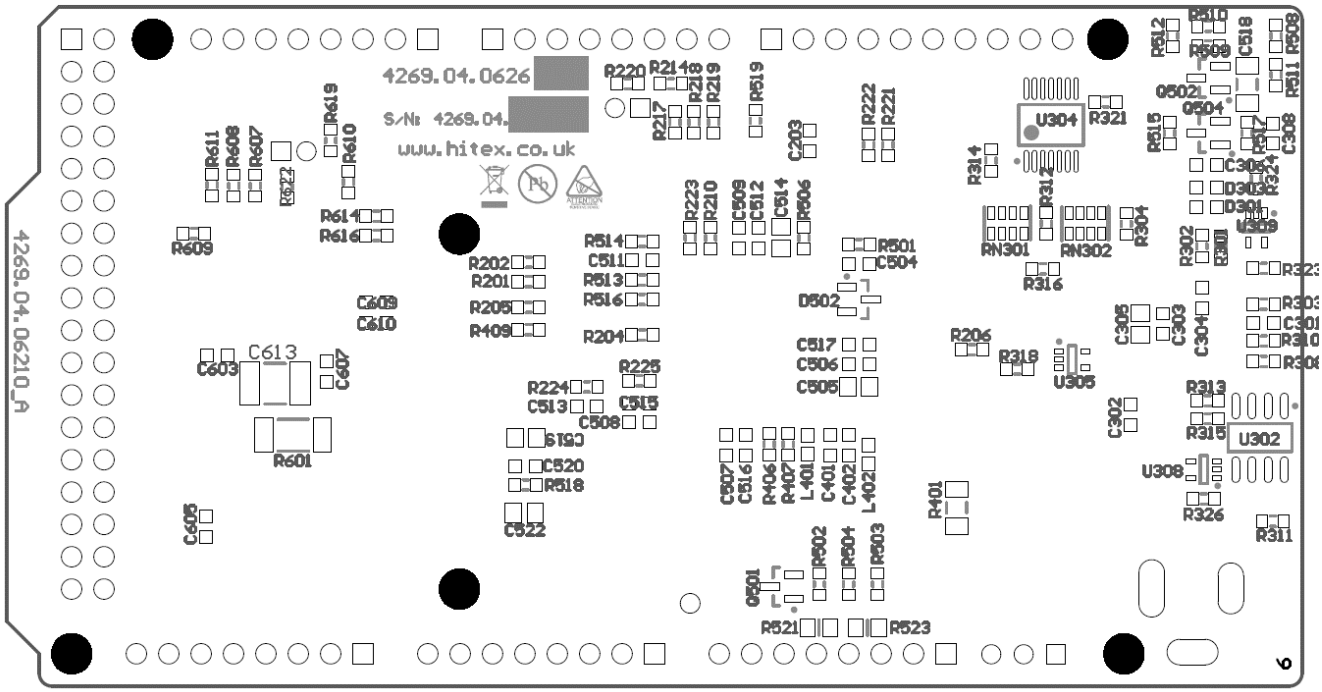


Figure 8 Component Location – Underside

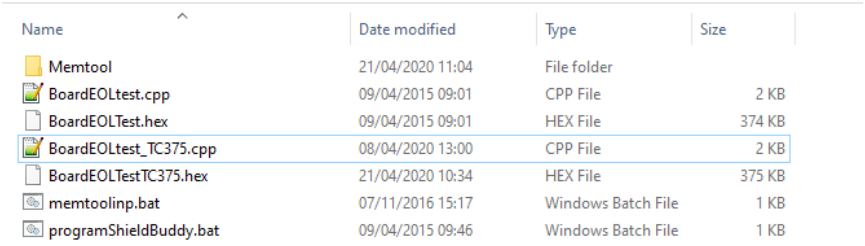
7 Appendices

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.

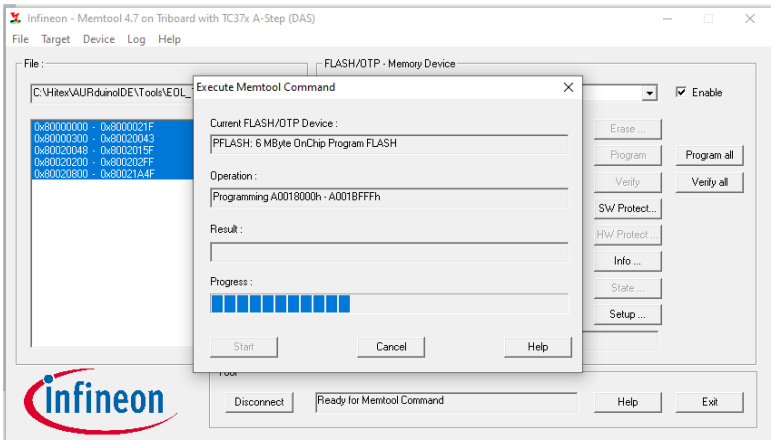
Go to “C:\Hitex\AURduinoIDE\Tools\EOL_Test”



Connect the ShieldBuddy to the USB port on your PC.

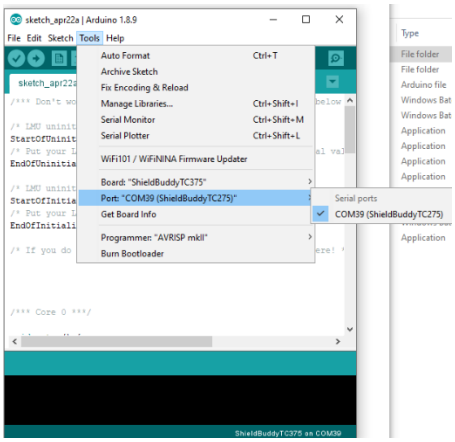
Wait for DAS to detect the ShieldBuddy – this takes about 15 seconds. Run the batch file “programShieldBuddy.bat”.

The Memtool programming tool will start and program the BoardEOLTestTC375.hex hexfile into the TC375 FLASH.

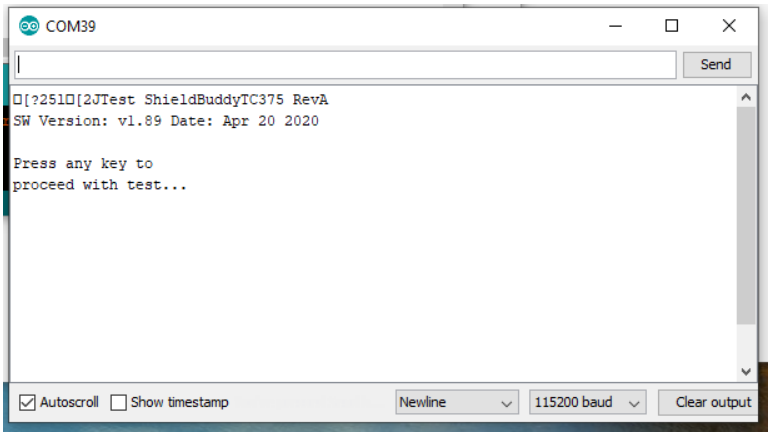


Now start a terminal emulation program (e.g. Arduino Serial Monitor, MTTY, Hyperterm etc.). The COM port created by the ShieldBuddy will vary from PC to PC. You can find it in the Arduino IDE under "Port:"

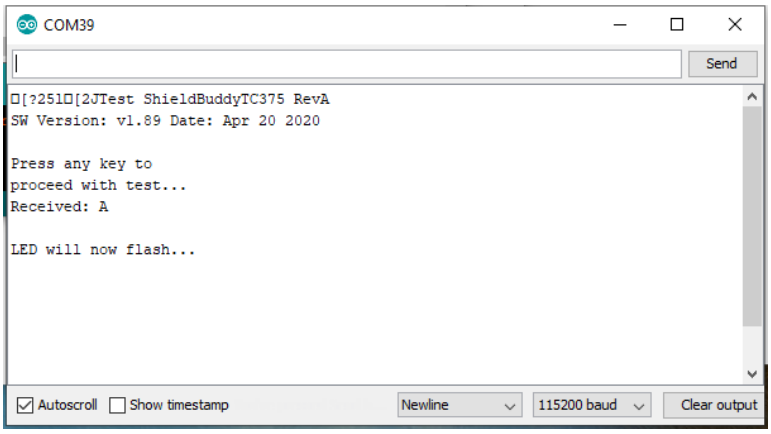
Use 115200 Baud.



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' and then Send. The key you pressed will be printed into the terminal and the LED on the ShieldBuddy pin 13 should start to flash. As the test program was designed for a VT100 terminal, some weird control characters will appear at the start.



That completes a successful test.

www.hitex.co.uk

Published by Hitex (U.K.) Limited.

4269.40100, 1.0