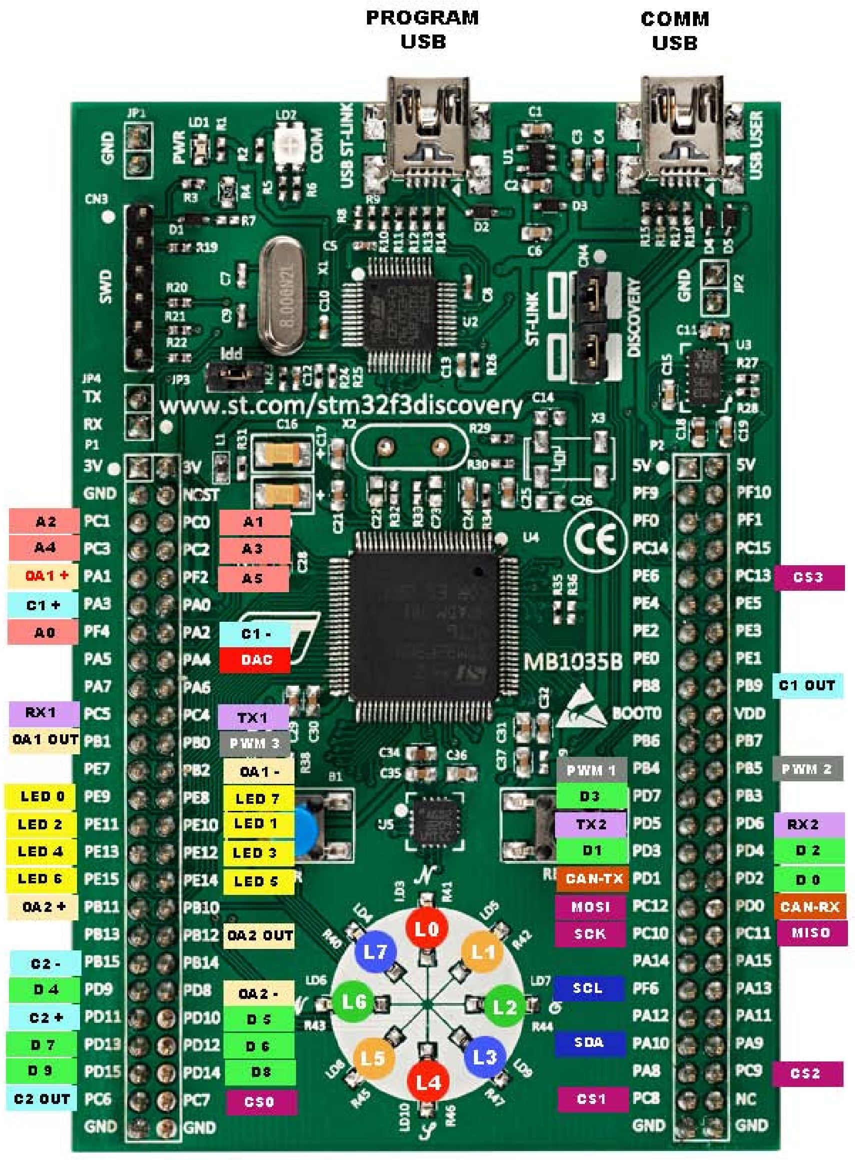
F3 Gizmo Manual



Version 1.0

Vicente Jiménez 25/4/2014

Index

DISCLAIMER AND LICENSE

Disclaimer

License

1. INTRODUCTION

The F3 Gizmo is a firmware solution for the STM32F3 Discovery board.
This board includes a lot of functionality for its price. Unfortunately it is not as easy to access to the board internals without some embedded knowledge and tools.
With the Gizmo you can access interactively to a lot of the board hardware features using a language that hides most of its complexities and using just an USB console interface.

The Gizmo works in RAM, so you can experiment as much as you want without needing to write the flash memory.

As the programming is interactive you have full access to the internal state of the system making debug easy.

The Gizmo supports threads, so you can run a program in a thread and debug is operation in another.

There are three main components in the Gizmo:

ChibiOS/RT
This is a real time operating system (RTOS). It provides the Gizmo with all the basic services it needs to operate. That includes board and CPU initialization a Hardware Abstraction Layer (HAL) and threading services.
If you want to develop in C using a STM Discovery Board ChibiOS/RT gives a solid foundation for your code.

CForth
This is a Forth like programming language. It provides the methods to interact with the Gizmo and program its actions.
CForth is interactive and its programming is RAM based. It is system independent and compiles in any 32 bit machine that supports the GNU C compiler.

F3 Gizmo PORT
This is a CForth extension that wraps around the ChibiOS/RT system services and some other services that directly interface with the STM32F303 hardware.
All Gizmo port CForth extensions except the ***MS*** word are not ANS standard words.

In order to be really interactive the programming in the Gizmo is done in RAM. As medium size microcontrollers have much more Flash than RAM memory, the Gizmo is designed to make an efficient use of RAM and to leave in Flash all elements that can be precompiled.

This manual only covers the CForth extensions to interface the hardware of the STM32F3 Discovery board. The general CForth operation is covered in an independent CForth manual. It is recommended to read the CForth manual before reading this manual.

2. GIZMO BASICS

2.1. Nomenclature

This subsection describes the basic nomenclature used in this manual.

n : Any number

u : Any number equal or greater than zero

c : A character (8 bit signed number)

f : A flag, that is, a number that can be interpreted as FALSE (zero) or TRUE (not zero)

addr : A number that points to one address inside the 32 bit memory space of the system CPU

Uaddr : A number that points to the offset of one memory element inside the User Dictionary (User Memory area)

<word> : A valid word name
Any valid sequence of characters that CForth understands

<uword> : A valir user word name
Any valid sequence of characters defined to perform a user defined functionality

WORD : One specific word name. Although CForth makes no case differences standard word names will be written uppercase to make the stand out.

`code` : Code written on the console or read from the console

Brown Text: Examples included in the manual

All F3 Gizmo CForth extension words are normal base dictionary words so they can be included inside word definitions.

2.2. Programming the firmware

The F3 Gizmo is a firmware solution. If you want to use it you need to flash it on a STM32F3 Discovery board. This board can be programmed using one of its USB connectors and a mini USB-B to USB-A cable like the one in the figure below.

STM provides a free STLink Utility application to program the Discovery boards under windows. You can currently find it in the reference below:

http://www.st.com/web/en/catalog/tools/PF258168

I know that the board can also be programmed under Linux using, for instance, Open OCD but I don't know the details of the installation. Son in this manual only the windows installation using STLink will be explained.

If you have a development environment for ARM Cortex MCUs under GCC you can also build all the system from its sources. If you don't have the time or the tools you can just use the provided F3Gizmo_Vxxx.bin binary file where xxx is the current version number.
For this manual we will considerer the following binary file:

F3Gizmo_V1.0(2014-04-20).bin

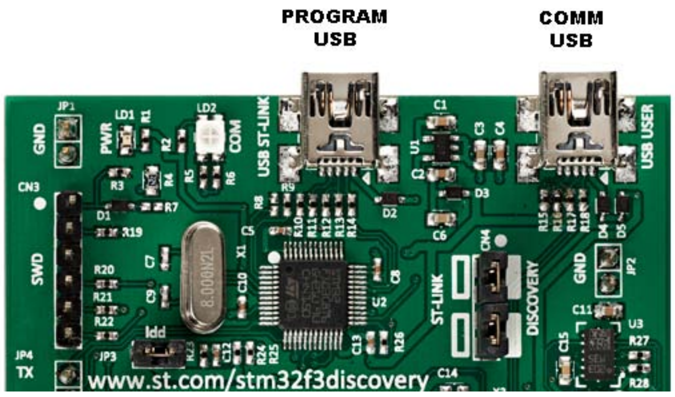
In the current F3 Gizmo version the filename could be different.

To install the firmware using this manual you need to have:

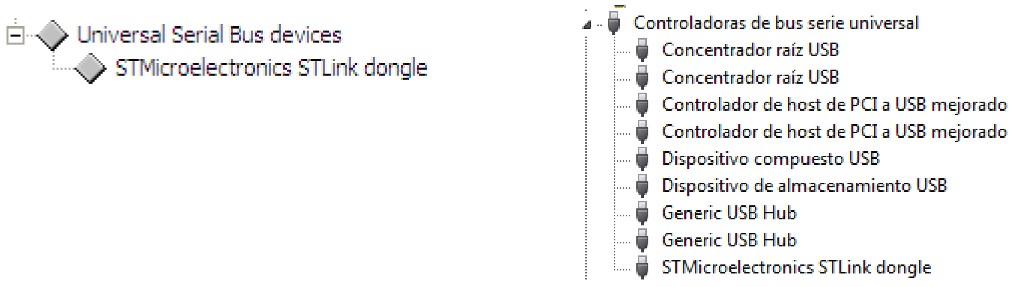
- A STM32F3 Discovery board
- A USB-A to USB-B micro cable
- STLink installed on a PC computer
- A binary firmware file with .bin extension

To install the firmware we must follow a sequence of steps:

1) Connect the STM32F3 Discovery board to the PC using the USB-A to USB-B micro cable. The board has two female USB connectors. We will call them the **Program USB** and the **Comm USB**. Use the **Program USB** one. That is, the one, centered in one board side as is shown in the figure below:

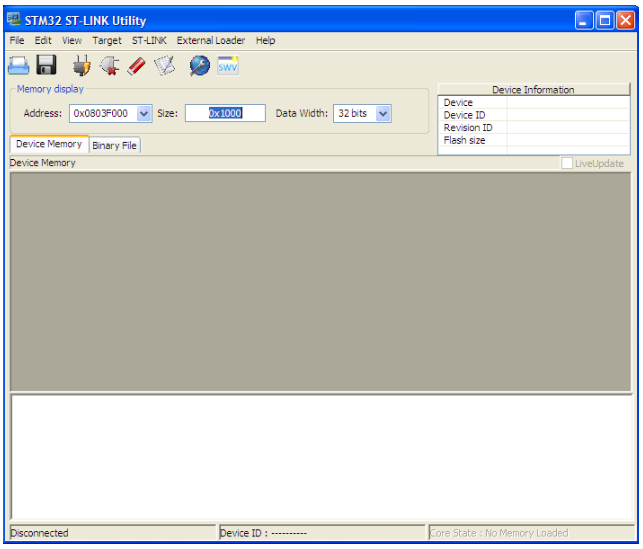


If you have correctly installed the ST LINK Utility that includes the necessary USB drivers the board should install and show in the device manager as a STLink dongle. The figure below shows how they are shown in Windows XP and Windows7:


Universal Serial Bus devices
STMicroelectronics STLink dongle

Controladoras de bus serie universal
Concentrador raíz USB
Concentrador raíz USB
Controlador de host de PCI a USB mejorado
Controlador de host de PCI a USB mejorado
Dispositivo compuesto USB
Dispositivo de almacenamiento USB
Generic USB Hub
Generic USB Hub
STMicroelectronics STLink dongle

..2) Launch the ***STM32 STLINK Utility***. You will get a window like the one in the following picture:

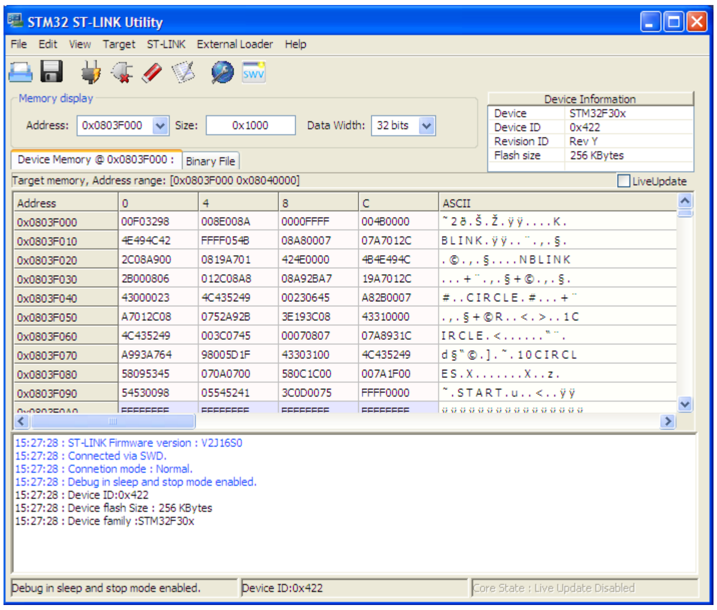


Address: 0x0803F000 Size: 0x1000 Data Width: 32 bits

Device
Device ID
Revision ID
Flash size

Disconnected | Device ID : --------- | Core State : No Memory Loaded

3) Connect with the board using the connect button. This is the one that looks like a plug. The device should identify as a STM32F30x device. In the STLink window you will see the current MCU contents.


STM32 ST-LINK Utility
File Edit View Target ST-LINK External Loader Help
SWV
Memory display
Device Memory @ 0x0803F000 :
Binary File
LiveUpdate

Address: 0x0803F000 Size: 0x1000 Data Width: 32 bits

Device Information	
Device	STM32F30x
Device ID	0x422
Revision ID	Rev Y
Flash size	256 KBytes

Target memory, Address range: [0x0803F000 0x08040000]

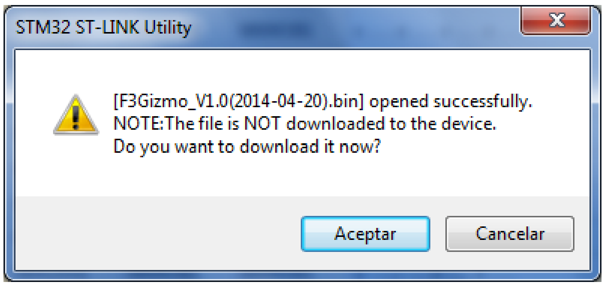
Address	0	4	8	C	ASCII
0x0803F000	00F03298	008E008A	0000FFFF	004B0000	˜2ð.Š.Ž.ÿÿ....K.
0x0803F010	4E494C42	FFFF054B	08A80007	07A7012C	BLINK.ÿÿ..¨.,.§.
0x0803F020	2C08A900	0819A701	424E0000	4B4E494C	.©.,.§....NBLINK
0x0803F030	2B000806	012C08A8	08A92BA7	19A7012C	...+¨.,.§+©.,.§.
0x0803F040	43000023	4C435249	00230645	A82B0007	#..CIRCLE.#...+¨
0x0803F050	A7012C08	0752A92B	3E193C08	43310000	.,.§+©R..<.>..1C
0x0803F060	4C435249	003C0745	00070807	07A8931C	IRCLE.<......"¨.
0x0803F070	A993A764	98005D1F	43303100	4C435249	d§"©.].˜.10CIRCL
0x0803F080	58095345	070A0700	580C1C00	007A1F00	ES.X.......X..z.
0x0803F090	54530098	05545241	3C0D0075	FFFF0000	˜.START.u..<..ÿÿ

```
15:27:28 : ST-LINK Firmware version : V2J16S0
15:27:28 : Connected via SWD.
15:27:28 : Connetion mode : Normal.
15:27:28 : Debug in sleep and stop mode enabled.
15:27:28 : Device ID:0x422
15:27:28 : Device flash Size : 256 KBytes
15:27:28 : Device family :STM32F30x
```

Debug in sleep and stop mode enabled. | Device ID:0x422 | Core State : Live Update Disabled

4) Load the F3 Gizmo firmware file using ***File*** → ***Open File***. In our example the F3Gizmo_V1.0(2014-04-20).bin file.

5) Transfer the firmware to the board.
As soon as you load the firmware bin file you a window should pop-up asking to download on the board. Just click the **Accept** button and then the **Program** button. Don't change the load address.



[F3Gizmo_V1.0(2014-04-20).bin] opened successfully.
NOTE:The file is NOT downloaded to the device.
Do you want to download it now?

If the previous window don't show up, you can manually download the file using

Target → ***Program & Verify..***

You should see in the STLink application a message like the following one that indicates that the firmware uploading has succeeded:

```
15:32:45 : [F3Gizmo_V1.0(2014-04-20).bin] opened successfully.
15:35:20 : Flash memory programmed in 8s and 187ms.
15:35:20 : Verification...OK
```

..6) Close the STLink application as it is no longer needed. Disconnect also the USB cable from the computer and the board.

Once you have flashed the firmware you can start using the Gizmo.

2.3. The console

The Gizmo interfaces the user through a console. There are three COM interfaces implemented by Gizmo on the STM32F3 Discovery board. Two serial TTL interfaces and one Serial over USB interface.
When the Gizmo starts it listens on both the Serial over USB and the Serial TTL channel #2 at 38400 baud. When the user enters a character with code 10 or 13 (just use the return key to do that) on one of these interfaces, the Gizmo selects it as its console. The other interface won't be available for the console until next reboot.

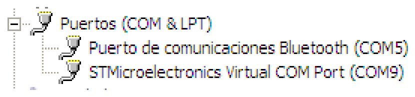
The console is the main user interface in the Gizmo. You send commands, program and get status information using the console.

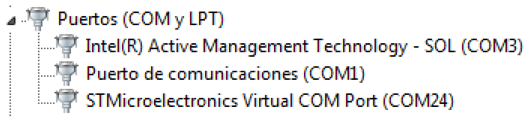
2.3.1. Serial over USB interface

The easiest way to connect the Gizmo to a PC is using the Serial over USB interface. Just connect the **Comm USB** (the most external female USB mini connector of the STM32F3 Discovery board) to a PC using the same USB-A to USB-B mini cable you used to program the board and follow the driver installation. Don't use the central **USB Program** connector that was used for writing the Gizmo Firmware as this connector is only useful for software development and firmware installation. The **Comm USB** connector also powers the board so you don't need any other power supply. The USB communication is also less prone to buffer underruns that the serial TTL interface so it is the best option if available.

After connecting the board to the PC using the **Comm USB** connector and after the driver installation the serial over USB show up on the device manager as a COM port.

The following figures show how they show up in the device manager as COM9 in a Windows XP machine and as a COM24 in a Windows 7 machine:


Puertos (COM & LPT)
Puerto de comunicaciones Bluetooth (COM5)
STMicroelectronics Virtual COM Port (COM9)


Puertos (COM y LPT)
Intel(R) Active Management Technology - SOL (COM3)
Puerto de comunicaciones (COM1)
STMicroelectronics Virtual COM Port (COM24)

2.3.2. Serial TTL interface

Serial Channel #2 can be used as a secondary alternative for the console. You can skip this section if you use the Serial over USB interface.

The serial channel #2 uses two GPIO lines of the Discovery Board:

PD5 : TX2 : Board Tx : Connect to PC Rx
PD6 : RX2 : Board Rx : Connect to PC Tx

Beware that you cannot directly connect these lines to a standard RS-232 PC Serial connector. The Discovery Board operates at 3V and the standard RS-232 features voltages at least between +/-7V. A direct connection of a standard RS-232 interface to the Discovery board would damage it and could damage both.

To use the Serial TTL Channel #2 to connect the Gizmo to a PC, you can, for instance:

a) Use a level converter that translates the +3V/0V signaling voltages of the Discovery Board to the -7V/+7V (or more) signaling voltages of the RS-232 standard.

b) Use a FTDI cable that connects to the board on one side using TTL serial signaling and to a USB connector on the PC on the other side.

As GPIO pins PD5 and PD6 are supposed to be 5V tolerant, it should be safe to use a 5V signaling on those pins. I have done that but it your case do it at your own risk. Beware that this is not true for the Serial Channel #1 that use GPIO pins that are not 5V tolerant.

Once you have solved the TTL serial connection you also need to power the board as the serial connection is not capable to do that. You can power the board using a mini USB-B to USB-A connector and a PC or a 5V supply. Read the STM32F3 Discovery manual to find about the different ways to power the board.

2.3.3. Using the console

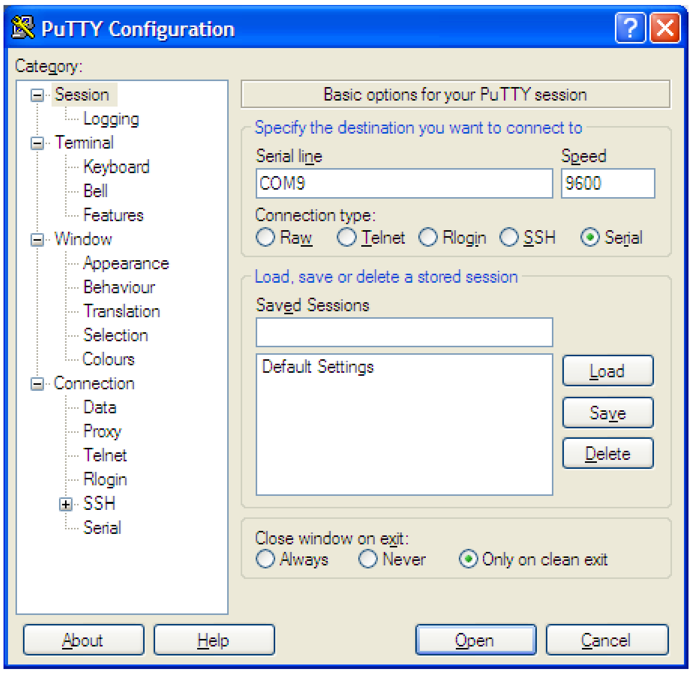
To access the F3 Gizmo console you will need a Terminal program like Hiperterminal or PuTTY. The examples in this manual will use the PuTTY program. You can find the download page for PuTTY at the end of this manual.
When the Gizmo starts it sends a message to both the Serial over USB and TTL Serial #2 to indicate that it is waiting for a connection.
If, for instance, you have a terminal connected to the Serial #2 channel you will get the following message:

```
    Gizmo: Serial #2 waiting for input...
```

A similar message will be sent also to the Serial over USB channel if present but you won't probably see it as the USB interface starts when you power the board and by the time you start the terminal the message is gone.

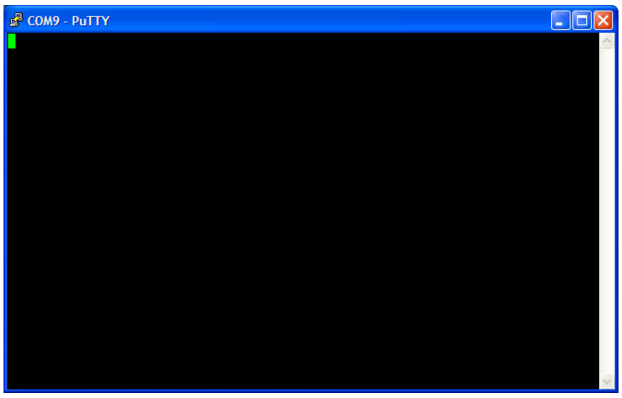
In this example we will show the use the USB connection as it is the easiest to use.

After the board **USB Comm** connector is connected to the PC using the cable and once you verify which is its associated COM port you can use PuTTY. Just call the **PUTTY.EXE** program, select serial connections and indicate the board comm port. As far as I know the selected speed is not important in serial over USB connections.

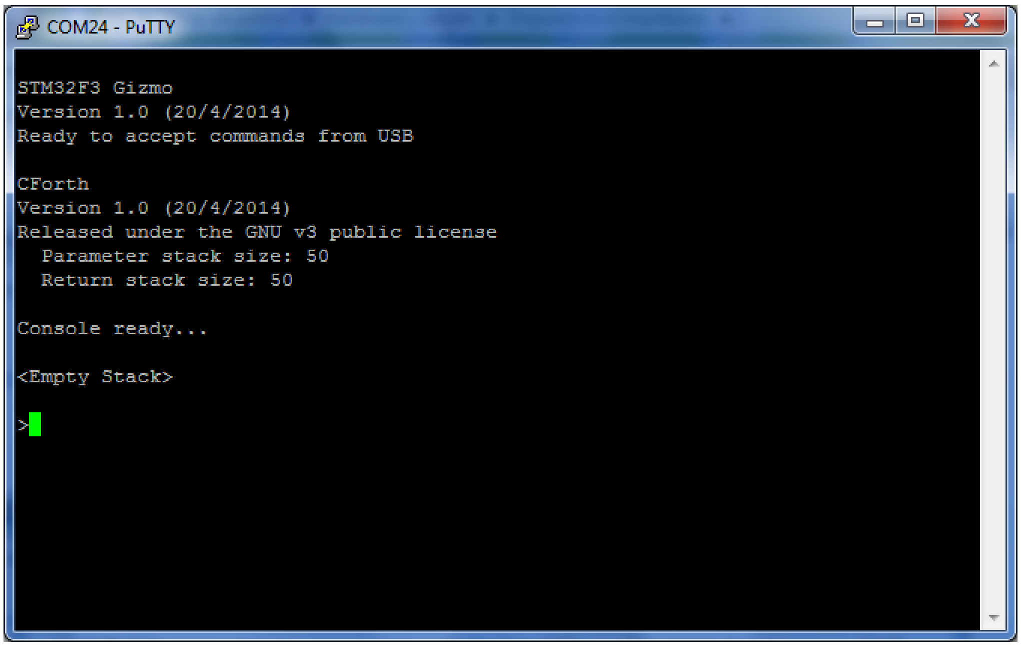


Serial line: COM9
Speed: 9600
Connection type: Raw Telnet Rlogin SSH [✓ Serial]
Close window on exit: Always Never [✓ Only on clean exit]

PuTTY should open a terminal. As it was explained before, the waiting for USB message is gone by the time you can open the terminal.



If you hit the return key the Gizmo will select the USB as its connection interface and acknowledge the received character with a start message like the following:



```
STM32F3 Gizmo
Version 1.0 (20/4/2014)
Ready to accept commands from USB

CForth
Version 1.0 (20/4/2014)
Released under the GNU v3 public license
  Parameter stack size: 50
  Return stack size: 50

Console ready...

<Empty Stack>

>
```

The Gizmo version (1.0 in the example) identifies the Gizmo specification. A change in version will probably add new functionalities and will be accompanied of a new version of this manual.
The date version (20/4/2014 in the example) identifies the date when the Firmware was built. Newer date versions for same Gizmo version identifies only bug fixes.
Observe that there are two version data lines.

As you can see the underlying CForth engine version is also shown. In this example the two versions, F3 Gizmo and CForth, are the same but it doesn't need to be.

The last character shown > is the prompt.

Congratulations, you can now start sending orders to the Gizmo.

The Gizmo console echoes your characters so it is better not to enable any echo on the terminal program.

You can tell from inside the Gizmo which console you are using by calling the ***CONSOLE?*** word.

CONSOLE? (-- u)
Indicates which is the current console.

0	No console
2	Serial#2 TTL console on PD5/PD6
10	Serial over USB console

If you use the ***CONSOLE?*** word when connected using the serial over USB connection you will get a 2 response.

```
>CONSOLE?
<1> Top: 10
```

Once you have the console running you can test everything from the CForth manual.
For the STM32F3 Discovery board specific CForth extensions keep reading.

2.4. CForth port implementation details

The F3 Gizmo implements CForth and adds some extensions over it. To know the current numeric limits in the F3 Gizmo use the ***LIMITS*** word. It will show the stack sizes and other CForth generic limits. It will also show specific Gizmo limits like the number of semaphores and mutexes.

2.4.1. PAD Implementation

To know the available memory in the User Dictionary call the ***UDATA*** word.

The F3 Gizmo uses the Core Coupled Memory (CCM) as the PAD space. As the CCM RAM in the STM32F303 MCU is 8kB we get 8192 Bytes of PAD space. That shows up using the ***UDATA*** word.
No current F3 Gizmo word uses the PAD so the data you put in the PAD will stay here while the board is powered. That could change in future F3 Gizmo versions.

2.4.2. Threads implementation

As it was previously explained the F3 Gizmo uses ChibiOS/RT as the underlying RTOS operating system.

Thread usage is explained in the CForth manual. This section only gives information on the thread priorities in the F3 Gizmo.

Thread priorities in this RTOS are absolute. That means that if there are several threads that can run, only the one with the higher priority will run. The console runs at priority 0 so any thread with higher priority will block the console.

As an example we can create a CPU hog word that consumes all available CPU time and run it at a priority over the one of the console:

```
: cpuhog BEGIN AGAIN ;
1 THPRIO cpuhog
```

That will block the console as it runs at higher priority and never blocks. To cancel this thread, and any other, push the USER button until you regain control of the console.

You can run the thread at the **same** priority of the console:

```
THREAD cpuhog
```

That won't block the console as threads with the same priority run in Round-Robin fashion alternating with each other. You can see that the thread is running using the ***TLIST*** word.

```
TLIST
 1 : CPUHOG Prio[0]  Running
```

To kill this thread use the ***TKILL*** word.

```
1 TKILL
Process [1] killed
```

You can also run process below the console priority:

```
-1 THPRIO cpuhog
```

As the console is waiting for user input most of the time, the **cpuhog** word will consume any available CPU time not consumed by the console.
Kill this thread before continuing.

```
1 tkill
  Process [1] killed
tlist
  No active threads
```

From all the above if you want be sure to give CPU time to all threads you should use the ***THREAD*** word that gives the same 0 priority to all threads. If you know what you are doing you can use the ***THPRIO*** word to set each thread priority but remember that only the thread at higher priority that can run will get the CPU time. The rest of the threads will run only when it blocks.

2.4.3. Memory implementation

The STM32F303 MCU includes 40kB of normal RAM and 8kB of CCM RAM. As we know the 8kB of CCM RAM is used for the PAD. The 40kB of normal RAM is shared between internal CForth and Gizmo operations and the User Dictionary.

The User Dictionary RAM space is allocated in 2kB pages. Version 1.0 of the F3 Gizmo uses 14 pages that give a total of 28672 Bytes. Some of the User Dictionary space is allocated for internal CForth use. That leaves 28652 Bytes left to the user. Enough for simple programs. For complex programs you should not being using the F3 Gizmo anyway.
Future F3 Gizmo versions could leave more or less RAM pages for the User Dictionary.

The F3 Gizmo uses the last pages of the MCU flash memory to save the User Dictionary by means of the ***SAVE*** word. If saved data is present in the flash memory when the board starts it gets loaded automatically. Remember that you can load Flash stored data anytime, eliminating any User Dictionary data from last boot, using the ***LOAD*** word.

2.5. Use of hardware lines

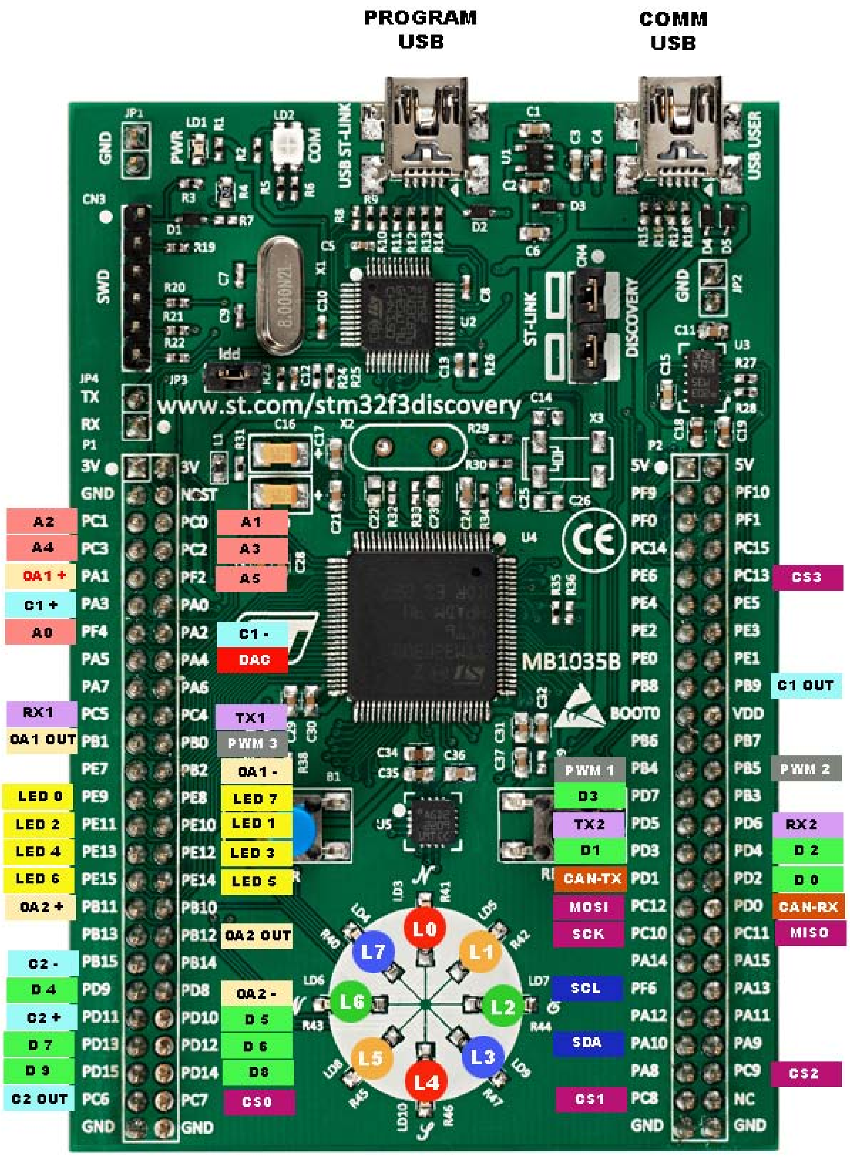
The STM32F3 Discovery is a very capable board. On the male headers to the left and right you can access all the microcontroller (MCU) pins. Each pin of the MCU can be used for several functions.

Changing the function of a pin can yield to incompatibilities with other pin functions and would make the Gizmo too complex to use. That's why the Gizmo uses a fixed pin mapping. Using a fixed mapping sacrifices versatility but if you need the maximum versatility the Discovery board can provide you won't be using the Gizmo anyway.

General use pins in MCU are usually named by its GPIO usage. GPIO stands for General Purpose Input and Output. In the MCU the GPIOs are grouped in 16 pin ports. Each port is related to a letter. On the STM32F303 MCU we have ports A to F although not all ports have all 16 lines implemented. ***PA7*** for instance means pin7 of port A.
The port name related to each line is indicated in the board on the white silkscreen.

In this section we list the allocated pin functions.

The following figure shows the current F3 Gizmo allocation of GPIO signals.



Some of the signals like LED, Analog and Digital lines are currently implemented in the F3 Gizmo while others like CAN, OA and CMP are reserved for future F3 Gizmo versions.

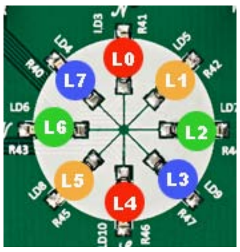
We will show now each F3 Gizmo functionality.

2.5.1. User Button (PA0)

This line is physically connected to the user button on the Discovery board. This is the blue button marked as *USER* in the silkscreen. In the Gizmo this line is linked to the program Abort function. If you push this button the ***abort*** flag will be set in all running Gizmo threads. That will force all running programs to terminate leaving only the main thread with the console running. It won't disable, however any active interrupt callback word.
If for any reason the board is irresponsive and this button don't help, you can use the board Reset black button that will restart the board. Beware that using Reset you will also erase any RAM contents not saved by the ***SAVE*** word.

2.5.2. User LEDs

The STM32F3 Discovery board includes 8 user LEDs positioned in a circular fashion.
We have numbered as 0 the led at the upper position (12 o'clock), From there we go following a clockwise sequence. Each LED is associated to one GPIO line in port E.



PE9	LED 0
PE10	LED 1
PE11	LED 2
PE12	LED 3
PE13	LED 4
PE14	LED 5
PE15	LED 6
PE8	LED 7

When a LED is activated the corresponding GPIO line is at high level (3V). When it is not activated the corresponding line is at low level (0V).
If you are out of digital lines you can use a LED line as a push pull Digital Output.

2.5.3. Digital I/O lines

In the F3 Gizmo 10 lines on port D are selected as Digital I/O lines. Each line can be configured in input and output mode. For input mode lines a pull up or pulldown resistor can also be selected. Output lines can be configured as push pull or open drain.

The Discovery board MCU is powered at 3V. Some of its pins are 5V tolerant but not all. You can damage the board if you use a greater than 3V voltage in a non 5V tolerant pin. Of course you should never apply more than 5V to any pin.

The following list indicates the lines that are used in the Gizmo for Digital I/O. Each line is numbered from 0 to 9.

PD2	D0	5V
PD3	D1	5V
PD4	D2	5V
PD7	D3	5V
PD9	D4	3V
PD10	D5	3V
PD12	D6	3V
PD13	D7	3V
PD14	D8	3V
PD15	D9	3V

2.5.4. PWM Lines

The Gizmo implements three PWM channels using the internal TIMER 3.

PB4	TIM3 Ch #1
PB5	TIM3 Ch #2
PB0	TIM3 Ch #3

2.5.5. Serial TTL Ch#1 / Ch#2

The Gizmo implements two serial channels. Channel #1 only can work at 3V. Channel #2 transmits at 3V but its input is 5V tolerant so it can probably used in 5V TTL serial connections.

PC4	TX1	3V
PC5	RX1	3V

PD5	TX2	5V	Console alternative
PD6	RX2	5V	

You should never attempt to connect a standard RS-232 serial line to the Discovery board. Standard RS-232 line levels are way beyond the limits that the board can tolerate.

Currently the Gizmo only uses serial #2 as an alternative for the console to the serial over USB connection. Future Gizmo versions will make more use of both serial channels.

2.5.6. Analog Inputs

The Gizmo has 6 pins reserved to analog inputs. These lines are associated to the inputs channels 5 to 10 of two ADC 12 bit converters.
In Gizmo the analog lines are numbered from A0 to A5 using the lines indicated below:

PF4	CH5	A0
PC0	CH6	A1
PC1	CH7	A2
PC2	CH8	A3
PC3	CH9	A4
PF2	CH10	A5

Each channel can be configured in single ended or differential mode. In differential mode the ADC converts the voltage difference between one channel and the next one.
The Gizmo supports reading one or two single ended or differential channels at the same time.

2.5.7. Analog Output (DAC)

The Gizmo supports one 12 bit analog output associated to the output of a **DAC**. It is associated to the PA4 line.

2.5.8. Serial buses

The F3 Gizmo uses two serial buses. One I2C bus and one SPI bus.

The I2C words manage the MCU I2C2 peripheral. It uses the following two lines for SCL and SDA. Both are 5V tolerant so you can use both 3V and 5V devices.

PF6	I2C2 SCL
PA10	I2C2 SDA

In order to use the I2C bus you must provide the two pull-up resistors.

The SPI words manage the MCU SPI3 peripheral. The bus uses three lines SCK, MISO and MOSI and an additional chip select line for each slave device.
Four lines are reserved for SPI devices. Beware that the last one is not 5V tolerant.

SPI Line	GPIO	
SCK	PC10	5V
MISO	PC11	5V
MOSI	PC12	5V

CS* Line	GPIO	
CS0*	PC7	5V
CS1*	PC8	5V
CS2*	PC9	5V
CS3*	PC13	3V

2.5.9. Gyroscope

The STM32F3 board includes a L3GD20 gyroscope that communicates with the MCU using the SPI1 peripheral that is independent from the one described in 2.5.8.
The following lines are reserved for this device.

PA5	SPI1 SCK
PA6	SPI1 MISO
PA7	SPI1 MOSI
PE0	INT1
PE1	INT2
PE3	CS*

The Gizmo provides functions to do basic readings of this device. For more complex operations it provides full access to the device internal registers.

2.5.10. Accelerometer / Compass

The STM32F3 board includes a LSM303DLHC combined Accelerometer/Compass that communicates with the MCU using the I2C1 peripheral that is independent from the one described in 2.5.8.
The following lines are reserved for this device.

PB6	I2C1 SCL
PB7	I2C1 SDA
PE2	DRDY
PE4	INT1
PE5	INT2

The Gizmo provides functions to do basic readings of this device. For more complex operations it provides full access to the device internal registers.

2.5.11. Reserved pins

The Gizmo is an ongoing project. There are several MCU pins that are not currently used but are reserved for future versions of the Gizmo. This last subsection describes the future F3 Gizmo functionalities.

Can Bus

Pins PD0 (RX) and PD1 (TX) are reserved for CAN Bus operation.

PD0	CAN RX
PD1	CAN TX

Comparators

The MCU includes seven comparators.
The following pins are reserved for the use of two of them:

CMP 1	PA3	INP	C1+
	PA2	INM6	C1-
	PB9	OUT	C1 OUT

CMP 2	PD11	INP	C2+
	PB15	INM7	C2-
	PC6	OUT	C2 OUT

Operational Amplifiers

The MCU includes four Operational Amplifiers.
The following pins are reserved for the use of two of them:

AO 1	PA1	INP	AO1 +
	PB2	INN	AO1 -
	PB1	OUT	AO1 Out

AO 2	PB11	INP	AO2 +
	PD8	INN	AO2 -
	PB12	OUT	AO2 Out

The following sections will give more details on the described hardware elements. They will also describe three CForth interface words.

2.6. Manual examples

This manual will present examples of the described CForth extension words. In order to use all examples the following hardware elements are needed.

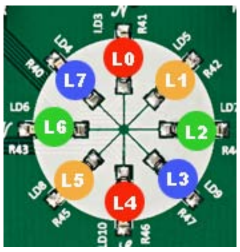
- STM32F3 Discovery Board
- USB-A to USB-B mini cable
- One small breadboard
- 1 Female to Female jumper cables
- 4 Female to Male jumper cables
- One Pushbutton
- Some 1/4W resistors: 330Ω, 3x1kΩ
- One LED diode
- An adjustable 1kΩ resistor

You can read the manual without trying the examples but using the STM32F3 board interactively while reading the manual is more fun and you will better learn its operation.

The example zones will be marked with brown color.

3. LEDS

As it was explained, the STM32F3 Discovery board includes 8 user LEDs. We number them from 0 to 7 in clockwise sequence. We rewrite here the port assignment for the LEDs to be used as a reference.



PE9	LED 0
PE10	LED 1
PE11	LED 2
PE12	LED 3
PE13	LED 4
PE14	LED 5
PE15	LED 6
PE8	LED 7

This section explain the CForth extension words that deal with the LEDs.

3.1. Working with LEDs one at a time

The LEDs are connected to 8 lines from line 8 to 15 of port E. Every port has two basic registers: The Output Data Register (ODR) is written to set the output values. If you read this register you read what was previously written and that don't necessary reflect the real status of the digital lines. The Input Data Register (IDR) is read to know the real status of the lines.
In the case of the LEDs, as they are output devices, we only deal with the ODR.

In this section ***u*** is a LED number that can be any number from 0 to 7.

LedSet `( u -- )`
Turns on LED ***u***

LedClear `( u -- )`
Turns off LED ***u***

The above words are the easiest to be used on the LEDs. Each one take a LED number from the stack and turns it on or off by modifying the pertinent ODR bit.

Try for yourself to turn on the first three leds.

```
0 LedSet
1 LedSet
2 LedSet
```

To turn off the LEDs:

```
0 LedClear
1 LedClear
2 LedClear
```

The two described words have a combination of uppercase and lowercase characters. As. CForth doesn't make differences in case you can write the words in uppercase, lowercase or any combination.
The words are shown, however with an exact case combination in all listings and help inside CForth.
Built-in words that include lowercase characters in the listings can be contracted using only the uppercase characters. So all the following expressions turn on LED 0:

```
0 LEDSET
0 LedSet
0 LS
0 ls
0 Ls
```

If you want to set the led status depending on a flag you can use the ***LedWrite*** word.

LedWrite `( f u -- )`
Set LED ***u*** depending on flag f. If f is 0 turns off the LED otherwise turns it on.

You can also read the LED status using the ***LedRead*** word.

LedRead `( u - f )`
Reads LED ***u*** related ODR bit.
If the LED is on, f wil be -1. If it's off f will be 0.

For instance you can toggle led 0 using:

```
0 LedRead
TRUE XOR
0 LedWrite
```

Or in a more compact way:

```
0 LR TRUE XOR 0 LW
```

You can define a word for this operation

```
: toggle0 0 LR TRUE XOR 0 LW ;
toggle0
```

3.2. Working with several LEDs at the same time

The previous words deal with one LED at a time. The following words can deal with all LEDs at the same time. To do that, each LED is associated to a bit in an 8 bit number. That way LED 0 has weight 1, LED 1 has weight 2 and so on till LED 7 that has weight 128. This way all LEDs information can be written in a number between 0 and 255.

In the rest of this section we will consider ***ub*** as a number between 0 and 255.

LedBinSet `( ub -- )`
Set each LED depending on the status of its corresponding bit.
For each bit of ***ub*** the corresponding LED is turned ON if it is "1".
Bits at "0" don't affect the corresponding LED status.

LedBinClear `( ub -- )`
Clear each LED depending on the status of its corresponding bit.
For each bit of ***ub*** the corresponding LED is turned OFF if it is "1".
Bits at "0" don't affect the corresponding LED status.

The following code turns ON and OFF all leds:

```
255 LedBinSet     \255 LBS can be also used
255 LedBinClear   \255 LBC can be also used
```

The words LedBinWrite and LedBinRead are the equivalent to LedWrite and LedRead for all LEDs at the same time.

LedBinWrite `( ub -- )`
Turns ON or OFF each LED depending on the status of its corresponding bit.
For each bit of ***ub*** the corresponding LED is turned ON if it is "1" and turned OFF it is "0".

LedBinRead `( -- ub )`
Read the status of all LEDs at the same time.
For each LED the corresponding bit of ***ub*** is set to "0” if the LED is OFF and set to "1" if the LED is ON.

For instance you can toggle all leds:

```
: toggleAll LBR TRUE XOR 255 AND LBW ;
toggleAll
```

4. DIGITAL I/O

LEDs are output only devices. The 10 digital I/O lines can be configured instead for input and output. We repeat here the 10 lines associated with the Digital I/Os.

PD2	D0	5V
PD3	D1	5V
PD4	D2	5V
PD7	D3	5V
PD9	D4	3V
PD10	D5	3V
PD12	D6	3V
PD13	D7	3V
PD14	D8	3V
PD15	D9	3V

4.1. Setting Digital I/O modes

Each line can be configured in 5 different states:

Input	Input
	Input with Pull-Up
	Input with Pull-Down
Output	Output Push Pull (Forces "1"s and "0"s)
	Output Open Drain (Forces only "0"s)

All 10 lines start in input mode and when set to output they start at low 0V level.

There are five words to set each possible state.
For the rest of this section ***u*** can be any number from 0 to 9.

DigitalInput `( u -- )`
Sets line ***u*** to digital input

DigitalPullUp `( u -- )`
Sets line ***u*** to input with pull-up

DigitalPullDown `( u -- )`
Sets line ***u*** to input with pull-down

DigitalOUtput `( u -- )`
Set digital line ***u*** as push-pull output

DigitalOpenDrain `( u -- )`
Set digital line ***u*** as open drain

4.2. Digital I/O in output mode

In output mode the words are similar to the ones used in the LEDs. They affect the Output Data Register (ODR).
If you use one of the digital output words in a line configured in input mode, the ODR register will be changed for the line although its effects won't be shown until the line is configured to output mode.

DigitalSet `( u -- )`
Set digital line ***u*** to high level.

DigitalClear `( u -- )`
Clear digital line ***u*** to low level.

DigitalWrite `( f u -- )`
Set digital line ***u*** depending on flag f. If f is 0 sets low level otherwise sets high level.

DigitalReadOutput `( u -- f )`
Read digital line ***u*** ODR status.
This word reads the output ODR register. It doesn’t read the real output state. It just reads the last output written data.

You can work with all 10 digital lines at the same time using a number that assigns a bit from 0 to 9 to each line. In the rest of this section ***ub*** is a number between 0 and 1023.

DigitalBinSet `( ub -- )`
Set each digital output depending on the status of its corresponding bit.
For each bit of ***ub*** the corresponding output is set at high level if it is "1".
Bits at "0" don't affect the corresponding line status.

DigitalBinClear `( ub -- )`
Clear each digital output depending on the status of its corresponding bit.
For each bit of ***ub*** the corresponding output is set at low level if it is "1".
Bits at "0" don't affect the corresponding line status.

DigitalBinWrite `( ub -- )`
Sets the level of each output depending on the status of its corresponding bit.
For each bit of ***ub*** the corresponding output is set to high level if it is "1" and set to low level it is "0".

DigitalBinReadOutput `( -- ub )`
Read the output status of all digital lines at the same time.
For each line the corresponding bit of ***ub*** is set to "0" if the corresponding line was set to low level and set to "1" if the line was set to high level.

4.2. Digital I/O in input mode

The words that follow are mainly used in input mode as they read the Input Data Register (IDR).
If you use one of the digital input words in a line configured in output mode, the IDR will reflect the real estate of the external lines that can be different to the output set value if an external device is forcing the line level. That last case is frequent in lines configured in open drain mode.

DigitalRead (u -- f)

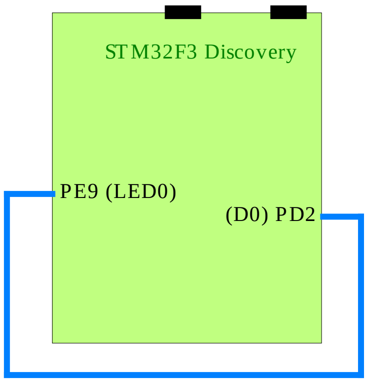
Reads the digital line ***u***. If the line is at low level it returns FALSE. If it is at high level it returns TRUE.

DigitalBinRead (-- ub)

Reads all 10 lines at the same time. For each line 0..10 the corresponding bit of ***ub*** is set to "0" if the line is at low level and set to "1" if the line is at high level.

4.4. Digital I/O examples

As an example of the use of the Digital I/O lines we will check one LED status using a Digital I/O line. We will connect PD2 associated to D0 to PE9 associated to LED 0 using a Female to Female jumper wire.



As Digital lines default to input mode we can check D0 status using digital read:

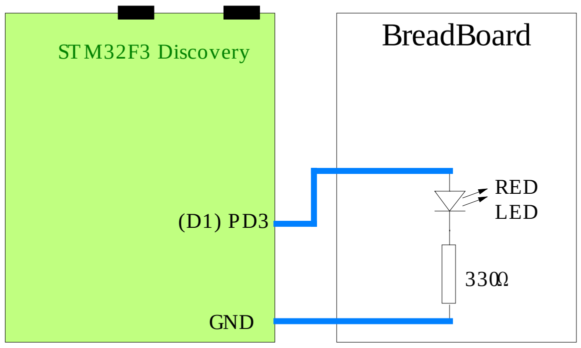
```
    0 DR
    <2> Top: 0
```

As we can see the response is 0 as LED 0 is OFF.
We can now turn LED 0 ON and check the digital line 0.

```
    0 LS
    0 DR
    <3> Top: -1
```

As we can see the response is -1 (Canonic TRUE) as LED 0 is ON and its line is High.

As a second example we can configure Digital Line 1 in output mode. We connect a series of a LED diode and a 330Ω resistor between the D1 line (PD3) and ground (GND).



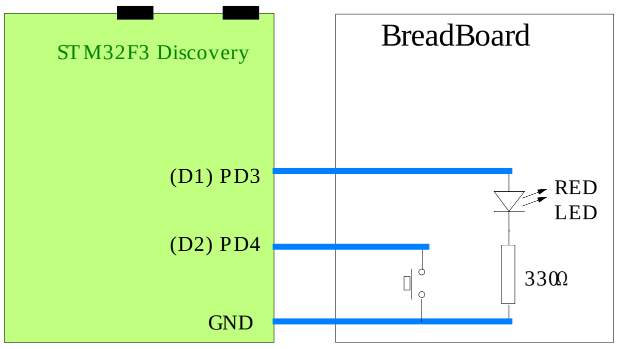
We first configure D1 as digital output and then we set it to high level.

```
1 DOU
1 DS
```

We should see that the LED turns ON. To turn it off:

```
1 DC
```

As a final example we will add a pushbutton between D2 (PD4) and ground.



To read the button it we will configure D2 as Digital Input with Pull-Up. That way D2 will read a High level when the button is not pushed and a Low level when it is pushed.

```
2 DPU
```

Then we will create a word that checks D2 and sets D1 to "1" if D2 is "0". As the button is active Low and the LED is active High we invert from the button read to the LED set.

```
: btest BEGIN 2 DR INVERT 1 DW AGAIN ;
btest
```

When the program runs it will light the led if we push the button.
To abort the program and regain console control we just need to push the blue USER button. An Abort message will be shown:

```
RUN ERROR: User Abort
Backtrace: 149 >> BTEST <<
```

As we know from CForth we can run the test program as a thread to be able to use the console while the program is running:

```
THREAD btest
```

To eliminate the thread we can list the current running threads and kill the one associated with the ***btest*** word.

```
TLIST
  1 : BTEST Prio[0]  Running

1 TKILL
Process [1] killed
```

5. PWM Channels

The F3 Gizmo uses one timer of the STM32F303 MCU, the TIMER3, to provide three hardware PWM channels.

PB4	TIM3 Ch #1
PB5	TIM3 Ch #2
PB0	TIM3 Ch #3

5.1. PWM Clock and Period configuration

All three PWM channels use a common clock source with frequency $\mathbf{f_{CK}}$. The clock is obtained from the 72MHz main MCU clock using a 16 bit prescaler. That limits the available frequency range for the timer clock $\mathbf{f_{CK}}$ from 1100 Hz to 72 MHz.

The clock drives a 16 bit counter whose maximum value $\mathbf{N_{PWM}}$ defines the PWW signal period $\mathbf{T_{PWM}}$. For instance, if you set the clock frequency $\mathbf{f_{CK}}$ to 10 kHz and maximum counter value $\mathbf{N_{PWM}}$ to 1000 clock cycles, then the PWM period $\mathbf{T_{PWM}}$ of all three channels will be 100 Hz.

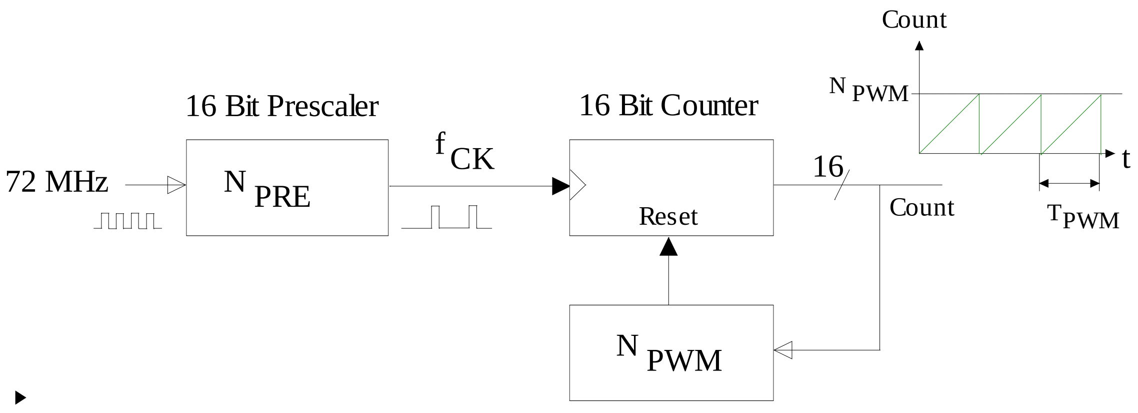
Putting it in algebraic form:

$$T_{PWM} = \frac{N_{PWM}}{f_{CK}}$$

We can also define a PWM signal frequency $\mathbf{f_{PWM}}$ and a clock period $\mathbf{T_{CK}}$ using the expressions:

$$f_{PWM} = \frac{1}{T_{PWM}} \qquad T_{CK} = \frac{1}{f_{CK}}$$

The following figure shows a representation of the PWM counter and its input clock circuit.



The 72 MHz MCU internal clock is fed to the programmable 16 bit prescaler. That prescaler generates one clock cycle for each N_{PRE} clock cycles generating the desired f_{CK} clock frequency. Each clock cycle the 16 bit counter is increased but it is reset to zero when it reaches the programmed N_{PWM} counter value.
That way the counter generates a sawtooth signal that goes from 0 to N_{PWM} one step each clock cycles but that returns to zero as soon as it reaches N_{PWM}.

To set the clock frequency and the PWM period we can use the following words:

PWMFreq `( uf -- )`
Set PWM clock frequency $\mathbf{f_{ck}}$ to ***uf*** (in Hz). This value must be between 1100 and 72 000 000.
The system automatically calculates and sets the necessary N_{PRE} value.
Changing the PWM frequency stops the operation of all three PWM channels.

PWMPeriod `( up -- )`

Set PWM period to ***up*** in clock cycles (**N_{PWM}**) at the selected clock frequency. This value must be between 1 and 65535. Using this word will affect the period of any current active PWM channel.

Observe that ***PWMFreq*** stops the PWM operation but ***PWMPeriod*** does not.

You can use the ***PWMSTOP*** word to stop all PWM operations.

PWMSTOP

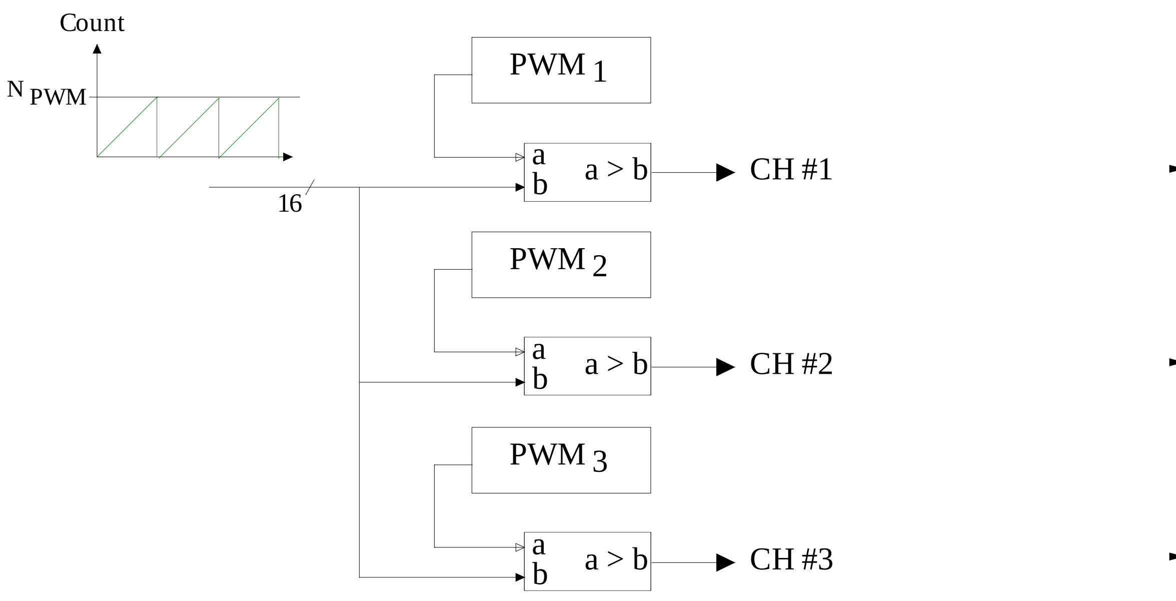
Stops the PWM system.
Leaves all channels in its current state.
Use Reset previously to guarantee low level in all channels.

For any desired PWM signal period there are usually several combinations of clock frequency and period. For instance to obtain a 10 Hz you can use f_{CK}=2000Hz and N_{PWM}=200 or you can use f_{CK}=10000Hz and N_{PWM}=1000.
In general it is desirable to have the biggest possible N_{PWM} value. That will provide the maximum PWM granularity. You must however take into account the limits that apply to f_{CK} and N_{PWM}. To obtain 10 Hz we could choose N_{PWM}=65535 and f_{CK}= 655350 Hz.
This is only one orientation, is up to you set the pair of N_{PWM} and f_{CK} values for any desired PWM period.

5.1. Setting a PWM channel

Each of the three PWM channels is connected to the 16 bit output of the counter. Each channel includes a PWM register so we have PWM_1, PWM_2 and PWM_3 for channels 1, 2 and 3. Each channel gives a High output value if the main counter is below its PWM_i value and gives a Low output value otherwise.

The following words activate and configure one of the three channels. For the rest of this section we will use ***uch*** as the number of channel that can be 1, 2 or 3.
The following figure shows the three channel PWM operation:



To control the PWM channels the following words can be used:

PWMSet `( up uch -- )`
Set ***up*** as the PWM$_i$ active number of cycles for channel ***uch***. This channel will be high during ***up*** clock cycles and low during the rest of the period. If ***up*** is higher that the period the channel will be at high level all the time.

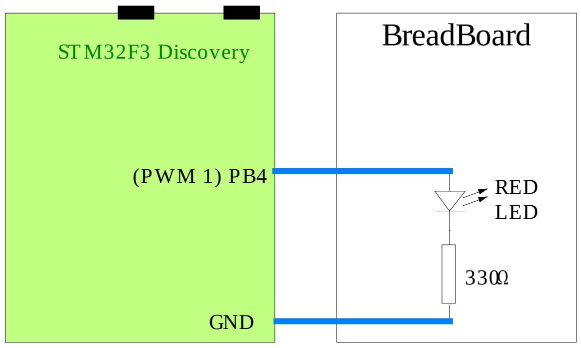
PWMReset `( uch -- )`
Deactivates one channel so it always gives a low output level.

5.3. PWM Examples

You can use to set the duty cycle of a LED. We can, for instance set a 10Hz PWM frequency f_{PWM} by selecting f_{CK}=100000 Hz and N_{PWM}=10000.

```
100000 PWMFreq
10000 PWMPeriod
```

We can connect a LED to Channel #1 by connecting the a series of a LED diode and a 330Ω resistor between PWM channel #1 output PB4 and GND.



The following code sets the duty cycle to 50% as it sets the PWM_1 value to half the PWM Period.

```
5000 1 PWMSet
```

To set a 10% Duty cycle and a 90% Duty cycle:

```
1000 1 PWMS
9000 1 PWMS
```

As the selected frequency is below the eye fusion frequency we can see the blinking. Using a 100 Hz frequency makes the eye only sensible to the mean value.

```
1000000 PWMF      \Set fck=1MHz
1000 1 PWMS       \Set 10% Duty
9000 1 PWMS       \Set 90% Duty
```

The following code is a more elaborated example. It creates a ***Pulse*** word that ramps up and down the intensity of a LED connected to the PWM Channel #1 at PB4.
As this is a long example you should probably cut and paste it on the terminal. Using PuTTY the Right Mouse Button pastes the clipboard.

```
1000000 PWMFreq    \1MHz fck
10000 PWMPeriod    \100Hz PWM Freq

: Pulse        \Slow blink program
0  { count }   \Count local val starts at 0
100 { inc }    \Inc local var starts at 100

BEGIN
  count 1 PWMSet    \Set PWM Channel #1
  inc +TO count     \Increase counter

  count 10000 > IF  \Check over high limit
      -100 TO inc
```

```
         10000 TO count
        ENDIF

     count 0 < IF      \Check below low limit
          100 TO inc
          0 TO count
        ENDIF

     10 MS             \Wait 10ms
   AGAIN ;             \Repeat the loop

   Pulse    \User Button cancels program
```

To end the example we can stop all PWM operations:

```
   PWMSTOP
```

6. TIME

The time words enable the user to use time controlled delays or to execute words an given time intervals.

6.1. Thread sleep

You can stop the current thread during a certain amount of time using the MS word.

> ***MS*** (u --)
> Wait the selected u number of milliseconds.

The ***MS*** word uses the **sleep** functionality that sleeps the current thread. In a multithreaded application other threads can run during this sleep time.

6.2. Hardware Timers

The F3 Gizmo uses two timers to provide time controlled execution of words.

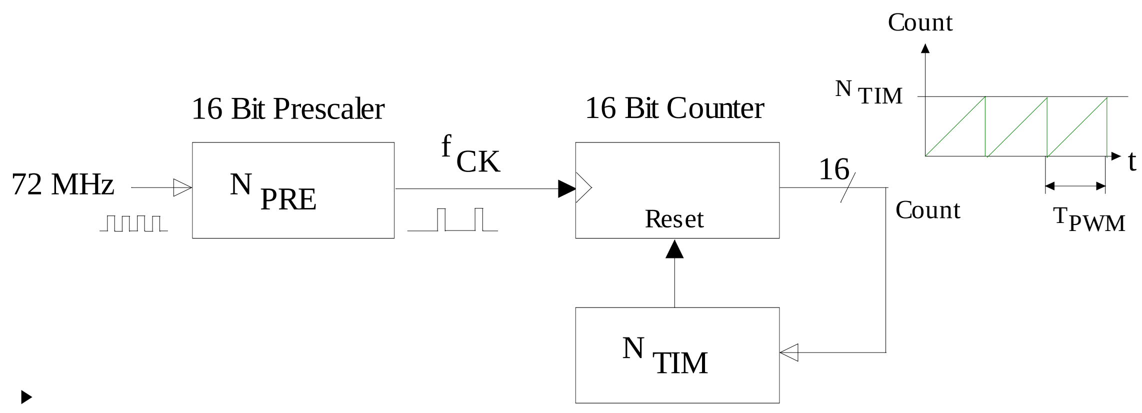
In the rest of this section ***ut*** will denote a timer number that can be 1 or 2. Both timers are independent of each other so all timer words need to specify the ***ut*** number.

6.2.1. Setting Timer Frequency

The frequency $\mathbf{f_{CK}}$ of each timer is set in a similar way that in the PWM case.

TimerFreq `( uf ut -- )`
Set the clock frequency $\mathbf{f_{CK}}$ of timer ***ut*** to ***uf*** (in Hz). This value must be between 1100 and 72000000.

The structure of each of the two timers is similar to the basic timer structure for the PWM channels:



Each timer can be used in three different ways. The easier use of one timer is a polled delay.

6.2.2. Polled Delay

The polled delay word stops the system until the selected timer counts a designated number of clock cycles.

TimerDelay `( ui ut -- )`
Polled delay of ***ui*** cycles for timer ***ut***.
Stops the system during ***ui*** cycles at the clock frequency $\mathbf{f_{CK}}$ of the ***ut*** timer.
Other threads cannot benefit of this pause. It is recommended to use the timer delay only for delays smaller than the minimum 1 millisecond that the MS word can provide.

6.2.3. Setting a Callback Word

The rest of functionalities that the timers provide use a **callback word**. A callback word is a word that is called when a timer expires.

TimerWord <word>   `( ut -- )`
Set the word callback of the timer ***ut***
If you use **NOWORD** as the word name no word will be assigned to this timer.

You cannot use the words ***FORGET***, ***FORGETALL*** or ***LOAD*** when there any callback word set. This prevents the timer to call a nonexistent word when it runs.
You cannot also call ***SAVE*** when there is a callback word set because an active timer could interfere with the saving process.

6.2.4. One Shot Timer

You can make the callback word to be executed in the future using the TimerOneShot word.

TimerOneShot   `( ui ut -- )`
Calls the callback after ***ui*** cycles of timer ***ut***.
The number of cycles must be between 1 and 65535.

That word configures the Timer ***ui*** Limit to ***ut*** and starts it. When the timer expires the callback word is called.

6.2.5. Periodic Call of a Word

You can also make the callback be executed peridiocally each certain time using the TimerRepeat word.

TimerRepeat `( ui ut -- )`
Calls the callback word every ***ui*** cycles of timer ***ut***.
The number of cycles must be between 1 and 65535.

That word configures the timer ***ut*** to an ***ui*** limit. Every time the limit is reached the callback word is called, the timer returns to zero and continues its operation.

To stop the repetition of callback calls you can use the ***TimerPause*** word.

TimerPause `( ut -- )`
Pauses the timer ***ut*** operation
Used specially for ***TimerRepeat*** cancelation

You can also use the ***TimerRESET*** word to pause all timers and removing all callback words at the same time.

TimerReset
Pauses all timers and removes all timer callback words.

This word can be uses to guarantee that it is safe to use, for instance, the ***FORGET*** word.

6.2.6. Timer Callback Context

The F3 Gizmo uses interrupts to call the callback word. As interrupt execution is asynchronous to normal run operation it must run in an independent context. The interrupt context contains a full context with parameter stack, return stack and verbose level.
Before the execution of the callback word the interrupt context parameter and return stacks are cleared and the verbose level is set to zero.

As in any other system it is recommended to keep interrupt execution time as small as possible. It is recommended for the callback word to only set flags and leave the main processing to a normal thread.

6.3. Time Examples

The ***MS*** word is the basic timing word in the F3 Gizmo. Using it you can develop a lot of time synchronized words.

The following example writes a countdown on the console:

```
\Countdown from 10 to 0
: countdown -1 10 -DO i . 500 ms -1 @loop ;
countdown
```

The following examples blink a LED. As both examples run forever you must stop them using the USER button.

```
\ LED 0 Blink
\ Blinks LED 0 forever
: blink begin 0 ls 300 ms 0 lc 300 ms again ;
blink

\ LED u Blink
\ Blinks LED u forever ( u -- u )
: nblink begin dup ls 300 ms dup lc 300 ms again ;
4 nblink
```

The following example lights all STM32F3 LEDs one at a time in clockwise order.

```
\ One circle
\ Light all LEDs in order one time
: 1circle 8 0 do i ls 100 ms i lc loop ;
1circle
```

The following toggle LED word is used to test the timer callback operation. It was used as a previous example so it is perhaps in memory. Check with ***UWORDS***.

```
\Toggle LED 0
: toggle0 0 LR TRUE XOR 0 LW ;
```

We set the timer 1 frequency to 10kHz and ***toggle0*** as the callback word.

```
\Set frequency
10000 1 TimerFreq

\Set word
1 TimerWord toggle0
```

The ***oneShot*** word toggles LED 0 and sets the timer in oneshot mode to change the LED agains after 5 seconds.

```
: oneShot toggle0 50000 1 TimerOneShot ;
oneShot
```

The ***forever*** word toggle LED 0 every second.

```
: forever 10000 1 TimerRepeat ;
forever
```

Observe that you cannot **abort** the timer using the USER button as it is not a running program but a periodic interrupt. As long as you press the USER button the toggle0 is aborted each time it is called and the LED doesn't change. But when you leave it the interrupt callback word returns to its normal operation.
To stop the timer you should use the ***TimerPause*** or ***TimerRESET*** words.

```
\Pause the timer
1 TimerPause
```

7. ANALOG

The Gizmo uses six of the GPIO lines of the STM32F3 Discovery board as analog input lines. In the Gizmo notation these lines are numbered from 0 to 5.

PF4	CH5	A0
PC0	CH6	A1
PC1	CH7	A2
PC2	CH8	A3
PC3	CH9	A4
PF2	CH10	A5

The STM32F303 microcontroller features several 12 bit Analog to Digital Converters (ADC) that can work in single ended or differential mode.

Each line from 0 to 4 can be configured in Single Ended or Differential mode. In Single Ended mode the voltage between the line and ground will be measured. In differential mode the voltage difference between one line and the next one is measured.
As there is no next line to line 5, this line cannot be selected as differential.

The MCU ADCs are 12 bit SAR converters. They can provide a 12 bit value between 0 and 4095. They use the 3V V_{DD} supply of the Discovery Board as a reference, so the converted value for each line in single ended mode is:

$$counts = 2^{12}\frac{V_{In}}{V_{DD}} = 4096\frac{V_{In}}{V_{DD}}$$

The count value is limited to values between 0 and 4095.

In Differential mode the measured value can range between $-V_{DD}$ and V_{DD}. As the range is doubled respect to the single ended mode the conversion function is different:

$$counts = 2^{11} + 2^{12}\frac{V_{In}^{+} - V_{In}^{-}}{2 \cdot V_{DD}} = 2048 + 4096\frac{V_{In}^{+} - V_{In}^{-}}{2 \cdot V_{DD}}$$

Observe that the converted value is not stored in 2's complement. ADCs use an offset representation where the 0 is at half the count range.

As in the case of the single ended mode the count value is limited between 0 and 4095. Differential mode readings have more dynamic range than single ones and are most useful for eliminating common mode noise.

The F3 Gizmo can read one Single Ended or Differential Channel at a time or, using two converters, it can read two Channels at the same time.

7.1. Basic analog readings

The STM32F303 converters are configured with an 18 MHz internal clock frequency, a sample time of about 182 clock cycles and a conversion time of 12 cycles. That means that the ADC uses about 11µs to do a conversion.

The following words configure each line as single or differential. By default all channels are configured as single ended.

AnalogSingle `( u -- )`
Set analog channel ***u*** to single ended. The channel ***u*** can be a number between 0 and 4.
Channel 5 cannot be selected as it cannot be configured in differential mode.
Channel 5 will be single ended if channel 4 is also single ended.

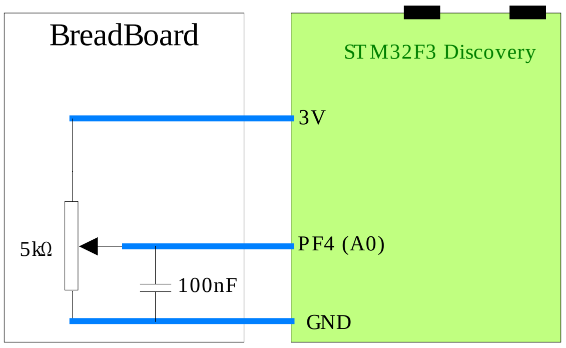
AnalogDiff `( u -- )`
Set analog channel ***u*** to differential where ***u*** is a channel number between 0 and 4. Setting channel ***u*** in differential mode makes channel u the positive input and channel u+1 the negative input.
If channel ***u*** is set to differential mode channel u+1 should not be used directly in analog read words.

To read an analog channel in single or differential mode the ***AnalogRead*** word can be used.

AnalogRead `( u -- ua )`
Read the analog value from channel ***u***
The ***ua*** read value can be between 0 and 4095 for input values between 0 and V_{DD} (in single ended mode) or between $-V_{DD}$ and V_{DD} (in differential mode).

As there are a lot or analog words we will insert the examples between word definitions.

To show the use of the analog read we can measure the voltage of a variable resistor cursor. In the image we show a 5kΩ value, but any value from 1kΩ to 50 kΩ should work.



The 100 nF capacitor is added to give more exact readings. That way the voltage won't drop during the ADC sample period.

We can define an ***a0leds*** word that lights one of the 8 board LEDs depending on the analog value read at A0.

```
\Set led depening on analog A0
: a0leds
BEGIN
 0 AnalogRead   \Read A0
 512 /          \Scale to 0..7
 255 LBC        \Clear all leds
 LS             \Set LED calculated before
 100 MS         \Wait 100 ms
AGAIN ;         \Repeat infinite loop

A0leds
```

To terminate the program abort it with the USER button.

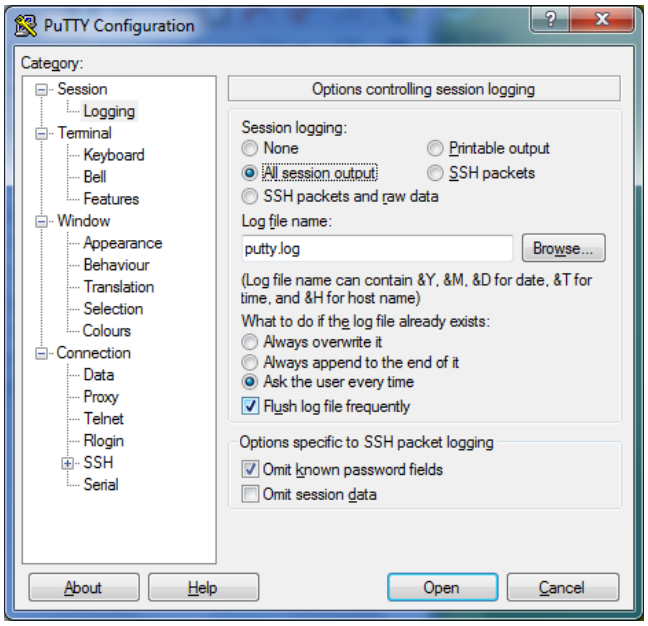
We can also run a program to give a reading on channel 0 every half second:

```
: readA0 0 AnalogRead . CR ;
: loopA0 BEGIN readA0 500 MS AGAIN ;

loopA0
```

As we can see there is some added noise because the readings are not all the same.

To better show the noise we can log the terminal data. To do that in PuTTY just close the terminal and reopen it enabling the log feature:



Options controlling session logging

Session logging:
( ) None ( ) Printable output
(●) All session output ( ) SSH packets
( ) SSH packets and raw data

Log file name: putty.log

(Log file name can contain &Y, &M, &D for date, &T for time, and &H for host name)

What to do if the log file already exists:
( ) Always overwrite it
( ) Always append to the end of it
(●) Ask the user every time

[✓] Flush log file frequently

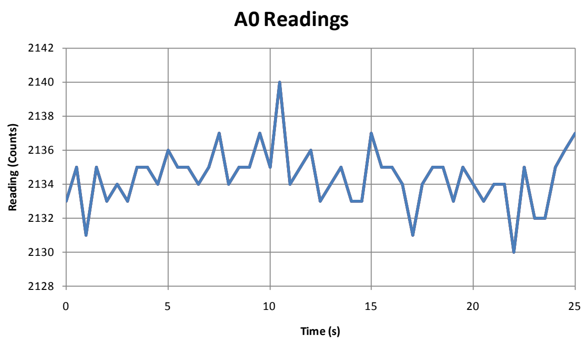
Options specific to SSH packet logging
[✓] Omit known password fields
[ ] Omit session data

The F3 Gizmo should work OK and continue just where you left it. If there is any problem just reboot the board.

We call the A0 read operation and stop with the USER button it when there is enough data.

```
loopA0
```

Use your favorite spreadsheet program to display the data:



7.2. Use of averaging to reduce noise

Each analog reading includes some added noise. You can reduce the measurement noise averaging several readings. To be precise you can only reduce the noise by averaging when it is not correlated to the signal and it has zero mean.
The follow two words enable the use of averaging in analog readings.

AnalogMean `( um -- )`
Set ***um*** to number of readings to average.
The ***um*** value must be between 1 and 500000.

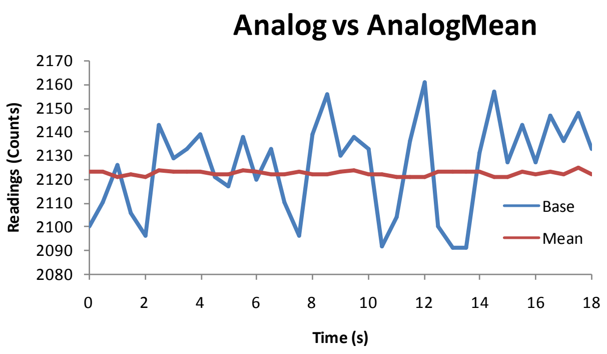
AnalogReadMean `( u -- ua )`
Reads the analog channel ***u*** the number of times selected with the ***AnalogMean*** word and provides an averaged value for all readings.

Using the same variable resistor example we can now provide normal and averaged values:

```
\ Read A0 in normal and mean mode
: loopA0m
."Base , Mean" CR
BEGIN
  0 AnalogRead . ." , "
  0 AnalogReadMean . CR
  500 MS
AGAIN ;

loopA0m
```

The averaged readings should be more close to each other than the base ones.



The mean value is around 2122. The measured 3V supply for the board is 2.973V. That means that the measured voltage is:

$$counts = 4096\frac{V_{In}}{V_{DD}} \qquad V_{In} = 2122\frac{2.973V}{4096} = 1.54V$$

Using a voltmeter I obtain a 1.541V value that is close enough.

7.3. Dual readings

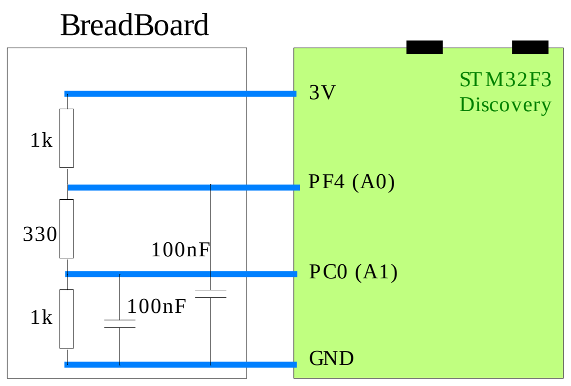
You can also perform two readings at the same time using two ADCs

AnalogRead2 `( u1 u2 -- ua1 ua2 )`
Read the analog value from channels ***u1*** and ***u2*** at the same time.
The ***ua1*** and ***ua2*** read values can be between 0 and 4095.

AnalogReadMean2 `( u1 u2 -- ua1 ua2 )`
Reads the analog channels ***u1*** and ***u2*** the number of times selected with the ***AnalogMean*** word and provides an averaged value for all readings.

Channel A0 can only be converted in ADC1, not in ADC2. So, if using A0 and another channel, A0 must be the first one. Trying to use A0 in ADC2 will give an error.

To test the dual and differential conversions we can use the following circuit:



You can read both channels at the same time:

```
0 1 AnalogRead2 .S CR
<2> 2337 1747
```

Observe that Channel A0 must be the first one:

```
1 0 AnalogRead2 .S CR
RUN ERROR: A0 cannot be used in ADC2
```

The exact values should be:

$$V_{In}(A0) = \frac{1330\Omega}{2330\Omega} V_{DD} \qquad counts_{A0} = 4096 \frac{V_{In}}{V_{DD}} = 2338$$

$$V_{In}(A1) = \frac{1000\Omega}{2330\Omega} V_{DD} \qquad counts_{A1} = 4096 \frac{V_{In}}{V_{DD}} = 1758$$

Close enough.
The difference between channels can be obtained by sustraction:

```
0 1 AnalogRead2 - . CR
589
```

The A0 to A1 difference can be also obtained using differential mode:

```
0 AnalogDiff     \Set A0 in Differential mode
0 AnalogRead     \Read A0
<1> Top: 2339
```

We can use the Differential mode formula to determine the ideal count value.

$$counts = 2048 + 4096 \frac{V(A0) - V(A1)}{2 \cdot V_{DD}} = 2338$$

Close enough again.

8.4. Conversion from counts to mV

All the analog read words provide count values between 0 and 4095. This is OK for ratiometric readings like, for instance, reading a potentiometer whose limits are connected to V_{DD} and GND. Other cases, however, need absolute voltage readings.

The follow two words can convert those values to voltage readings in milivolts.

Single2mV `( ua -- mV )`
Convert single reading to mV
ua is a 12 bit converted value between 0 and 4095
mV is the converted mV value between 0 and about 3000

Diff2mV `( ua -- mV )`
Converts differential reading to mV
ua is a 12 bit converted value between 0 and 4095
mV is the converted mV value between about -3000 and about 3000

To use the previous conversion words we must first calibrate the ADC readings using the ***AnalogCal*** word.

We can now repeat the previous readings. We first calibrate the ADC conversion:

```
AnalogCal
<1> Top: 2920
```

The previous word tries to calculate V_{DD} to calibrate the ADC values. In this case the 3V supply is read as 2,92V.

Remember that we had a resistance divider. Using the given V_{DD} value the voltage between A0 and A1 should be:

$$V(A0) - V(A1) = \frac{330\Omega}{2330\Omega} V_{DD} = 413\ mV$$

We can measure and show that value on screen in single ended mode:

```
0 AnalogSingle
0 1 AnalogRead2 - Single2mV
<3> Top: 419
```

Or using differential mode:

```
0 AnalogDiff
0 AnalogRead Diff2mV
<1> Top: 414
```

If we measure with a multimeter the real value of the voltage V(A0) - V(A1) in the 330Ω resistor we will read 424mV. The previous values were a little off. Usually this error is negligible but when you need the best available precision you need to calibrate.

8.5. Calibrating the converter

To obtain good absolute voltage readings, the value of the V_{DD} supply voltage needs to be known with enough precision. The problem is that it cannot be guaranteed an exact V_{DD} voltage. Let alone be it exactly 3V.

To determine the value of the V_{DD} supply, the MCU incorporates a 1.2V voltage reference connected to one of the converter channels. Reading this reference we can deduce the current V_{DD} value. We know that in single ended mode:

$$counts = 4096 \frac{V_{In}}{V_{DD}}$$

If we know that V_{In} = 1.2V we can deduce V_{DD}

$$V_{DD} = 1.2V \frac{4096}{counts}$$

The previously used ***AnalogCal*** word is used to calibrate the ADC by measuring the reference voltage.

AnalogCal `( -- mV )`
Calibrate the ADC. Masures the internal 1.2V voltage reference and uses it to calculate the Vdd supply voltage. Returns in the TOS the Vdd voltage in mV.

If we want we can measure the voltage reference channel using the ***AnalogReadRef*** word to do the calculations ourselves but it is not normally needed.

AnalogReadRef `( -- ua )`
Read the reference voltage (in 12 bit counts from 0 to 4095) using the ADC.

If we use this word we can obtain the count value of the internal reference measurement:

```
AnalogReadRef
<2> Top: 1680
```

Using the above formula:

$$V_{DD} = 1.2V\frac{4096}{counts} = 1.2V\frac{4096}{1680} = 2.926V$$

All this calculations are automatically done calling ***AnalogCal***:

```
AnalogCal
<3> Top: 2925
```

The use of the voltage reference reading improves the precision of the ADC readings but it is not perfect. This is due to the fact that the reference itself although good is not perfect. The datasheet for the STM32F303 MCU states that this reference can have an output voltage between 1.16V and 1.25V. That means that we can have an error between -3.3% and 4.1% in our readings.

There are no more references in the MCU so, if we want the maximum precision we must use an external measurement to calibrate the reference.
The easiest way to calibrate the reference is to measure the ***real*** V_{DD} voltage using a good enough multimeter. That way we can check the real value of the internal reference.
The real value of the internal reference, once determined, is stored in a special **port** register of the User Dictionary so it will be used in every following ***AnalogCal*** use and, if the User Dictionary is saved, it would be used for calibration when the User Dictionary is loaded.

AnalogVdd `( mV -- mV )`
Calculates V_{REF} from measured V_{DD}.
Before calling this words the real measured Vdd (in mV) must be pushed on the stack as the TOS. After calling the word the TOS will be substituted by the real value of the internal reference that should be between 1160 mV and 1250 mV.
This word updates the calibrated reference value in the User Dictionary.

Using a multimeter to check the voltage at the 3V pin of the STM32F3 Discovery board I read a 2.974V value. We can use this value to calibrate the internal V_{REF} value:

```
2974 AnalogVdd
<4> Top: 1221
```

So we know that the internal reference is 1.221V and not 1.2V. This value is now stored inside the F3 Gizmo so from now the read values will be more exact.

```
0 AnalogDiff
0 AnalogRead Diff2mV
<1> Top: 424
```

That is exactly the same 424mV I measured in the 330Ω resistor before using the multimeter. Using ***SAVE*** we save all User Dictionary data together with the V_{REF} calibration. The F3 Gizmo however will refuse to save the dictionary if you have not created any word.

```
SAVE

Unlocking flash
Erasing flash
Erasing page 126
Writing flash

Flash saving ended
```

If you know from a previous reading the value of the internal reference voltage and it was not saved in the user dictionary, you can set it manually using the AnalogVref word.

AnalogVref `( mV -- mV )`
Calculates Vdd from known Vref.
Before calling this words the real known V_{REF} (in mV) must be pushed on the stack as the TOS. After calling the word the TOS will be substituted by the measured value of V_{DD} using this V_{REF} value.
This word updates the calibrated reference value in the User Dictionary.

8.6. Measure the MCU temperature

The MCU includes a temperature sensor connected to one converter channel. This measurement has a very big offset error so its readings can only be used to check temperature variations, not absolute values.

AnalogReadTemp `( -- ut )`
Read MCU internal temperature ut in 10*C. That is, for 20°C we will read 200.

For example:

```
    ART    \We use the contracted name
    <1> Top: 323
```

That means 32.3ºC

8.7. Using the DAC

The F3 Gizmo uses one Digital to Analog Converter (DAC) at the PA4 pin that provides a 12 bit precision controlled analog voltage output. Its conversion function is complementary to the one of the ADC in single ended mode:

$$V_{OUT} = V_{DD} \frac{count}{4096}$$

We use the ***AnalogWrite*** word to set the DAC count value.

AnalogWrite `( ua -- )` Sets the DAC value between 0 and about V_{DD} from the counts ua value that can be between 0 and 4095.	

In fact the DAC cannot generate V_{DD} as the maximum 4095 count is one LSB below V_{DD}.

To generate an absolute output voltage we can use the mV2Single word:

mV2Single `( mV -- ua )` Converts mV to single reading counts	

The use of this word uses and benefits from the previously described ADC calibration.

For example we can set a 1.0V value (1000 mV) at the DAC output:

```
    1000 mV2Single AnalogWrite
```

In my multimeter that gives a read of 1.000V between PA4 and GND.

As we can see the performance of the ADC and DAC of the board can greatly improve by calibration. Of course you would not get any better precision that the one provided by the multimeter you use to calibrate the system.

9. SERIAL BUSES

As we explained before the F3 Gizmo includes two serial buses to connect to external devices. One SPI bus and one I2C bus.

9.1. SPI

The SPI commands use the SPI #3 peripheral of the microcontroller. All these commands use the Gizmo as the bus master. The Gizmo uses the clock and two data lines of the SPI bus and four chip select signals.

SPI Line	GPIO	
SCK	PC10	5V
MISO	PC11	5V
MOSI	PC12	5V

CS* Line	GPIO	
CS0*	PC7	5V
CS1*	PC8	5V
CS2*	PC9	5V
CS3*	PC13	3V

As each SPI peripheral needs to use one CS* line (at low level when active), the basic SPI interface provided by the Gizmo enables it to work with up to four peripherals. Using digital I/O lines (or LED lines) it is possible to work with more than four lines.

9.1.1. SPI Configuration

In order to configure a SPI connection, we must set the SPI mode and frequency. The SPIMode and SFreq commands are used to do make that configuration.

SPIMode `( u -- )`	
Set SPI mode between 0 and 3	

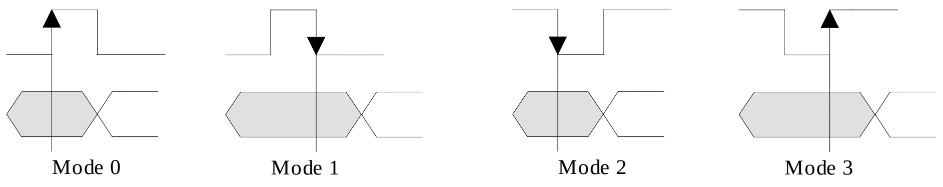
The SPI mode depends on two configuration parameters: clock polarity (cpol) and clock phase (cpha). Each of these parameters can have a value of 0 or 1. The SPI mode determines the configuration from both values.

Mode	cpol	cpha
0	0	0
1	0	1
2	1	0
3	1	1

The clock polarity cpol indicates which is the level of the clock line when it is not active. When cpol=0, clock line is at Low "0" level when inactive. When cpol=1, clock line is at High “1” level when inactive.

The clock phase cpha indicates which is the clock transition that synchronizes the data. When cpha=0, the data is synchronized at the first transition (from inactive state). When cpha=1, the data is synchronized at the second transition (from active state).

The next figure shows the first synchronized data for each SPI mode.



Once determined the SPI mode of operation, we must choose the frequency.

SPIFreq `( ukHz -- ukHz )`
Sets SPI Bus frequency
The desired frequency of operation (in kHz) is popped from the stack. The Gizmo tries to select the maximum available frequency of operation that is below the desired value and selects it for the SPI peripheral. That selected frequency is then pushed on the stack.

The SPI peripheral can currently work at eight frequencies between 141kHz and 18 MHz. If the desired frequency is below 141kHz the Gizmo will give an error message and select the minimum 141kHz frequency.

SPI transfers must start with the selection of one slave and must end with the deselection of the active slave.

SPIStart `( u -- )`
Select the slave u between 0 and 3

SPIEnd
Ends SPI transmission deselecting the active slave.

If we want to use more than four slaves we can use any I/O or LED line to access the slave CS* input. In that case we must activate the slave (setting a Low level in the CS* line) after calling the ***SPIStart*** word with a dummy slave line (0..3) and we must deactivate the slave (setting a High level in the CS* line) before calling the ***SPIEnd***.

9.1.2. SPI Transfers

The unit of transfer on the SPI bus is a byte (8 bits). The F3 Gizmo threats all transfers as unsigned so every transfered byte can have a value between 0 and 255.
The SPI bus is full duplex, for every transfer one byte is transferred from the master (STM32F3 Board) to the slave and one byte is transferred from the slave to the master.

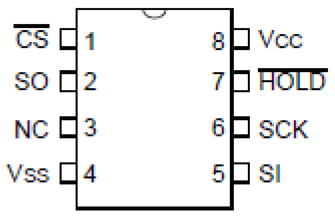
There are two words to provide a SPI transfer, one for only one byte and another for a series of bytes.

SPIByte `( umaster -- uslave )`
Interchanges one byte between the master and the slave. Before calling this word a byte to transfer from the master to the slave have to be pushed as TOS on the stack. After the transfer the TOS holds the slave response byte.

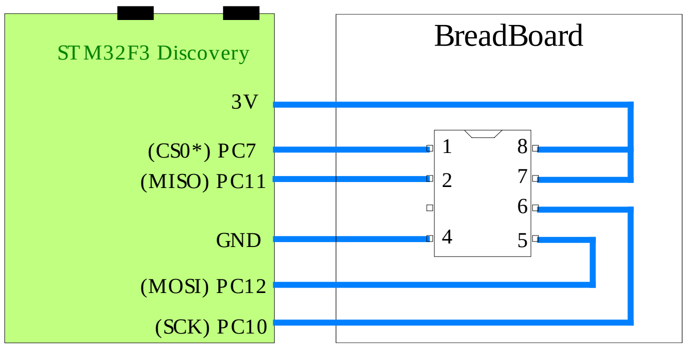
SPITransfer `( um1..umn un -- us1..usn )`
Interchanges ***un*** bytes between the master and the slave. The first byte sent from the master is ***um1*** and the last one ***umn***. For each byte sent by the master one byte is received from the slave. The received byte when ***um1*** is sent is ***us1*** and so on.

9.1.3. SPI Memory Example

To demonstrate the use of the SPI bus a Microchip 23K640 8k Byte SRAM SPI memory will be written and read. This device can be bought in an 8 pin DIP package that is very easy to use on a breadboard.



We make the necessary connections between the memory chip and the STM32F3 board:

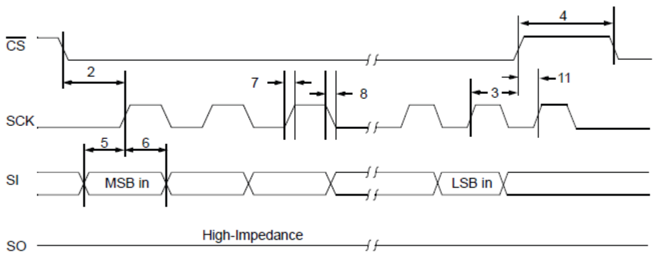


The datasheet states that the chip can work at least up to 16MHz when powered at 3V. We will set a 10MHz desired frequency:

```
\Set 10MHz frequency
SPIFreq 10000
<3> Top: 10000
```

As we can see 10MHz is one of the 8 working frequencies for the SPI bus.

From the datasheet we can find the signal timings:



As clock inactive state is Low and it syncs on rising edge the mode is 0.

```
\Set mode 0
0 SPIMode
```

The memory has 8k Bytes of space. That is 2^{13} Bytes between 0x0000 and 0x1FFF.
To write data on the memory we must send:

0x02 0xAH 0xAL 0xDD

Where:	0x02	Command to set write
	0xAH	High byte of address
	0xAL	Low byte of address
	0xDD	Data to write at address

To write a 0xAA value at position 0x03D0 we send:

0x02 0x03 0xD0 0xAA

Using the SPI bus words:

```
\ Write 0xAA at address 0x03D0
clear                   \Clear stack
0 SPIStart              \Activate slave
0x02 0x03 0xD0 0xAA     \Data to send
4 SPITransfer           \Transfer of 4 bytes of data
SPIEnd                  \Deactivate slave
.S                      \See what we get

<4> 0 0 0 0
```

For each Byte sent the slave responds with another byte. As the slave doesn't have any response to send it just sends four dummy 0 Bytes.

To read the 0x03D0 position we should use the read command:

0x03 0xAH 0xAL 0x00

Where:	0x03	Command to set read
	0xAH	High byte of address
	0xAL	Low byte of address
	0x00	Dummy data (Can be any value)

So the data to send is:

0x03 0x03 0xD0 0x00

Using SPI words:

```
\ Read at address 0x03D0
clear                  \Clear stack
0 SPIStart             \Activate slave
0x03 0x03 0xD0 0x00    \Data to send
4 SPITransfer          \Transfer of data
SPIEnd                 \Deactivate slave
.S                     \See what we get

<4> 0 0 0 170
```

The first 3 Bytes are dummy responses to command and address. The last 170 value is the response from the memory to our request. Is it the 0xAA we previously wrote?

```
X.   \Get TOS in hexadecimal
AA <3> Top: 0
```

Success!!

9.2. I2C

The I2C commands use the I2C #2 peripheral of the microcontroller. In a similar way that the SPI case, all these commands use the Gizmo as the bus master. The Gizmo uses a clock SCL and data line SDA for this bus. The pull-up resistors needed to use this bus are not included on the Discovery Board so you have to provide them yourself. In general you can use high R values for low speeds and low R values for high speeds.

The SPI bus needed 3 signals plus one CS* for each slave. The I2C bus only uses two signals for any number of slaves and can host more than a master on the bus.
The port mapping used for this peripheral is shown on the next table.

PF6	I2C2 SCL
PA10	I2C2 SDA

The I2C bus can contain several devices identified by an address number. The Gizmo currently only supports I2C devices with 7 bit addresses. In the 7 bit address group only the addresses between 0x08 and 0x77 are valid addresses. The rest of possible values are reserved for several I2C functionalities.

9.2.1. I2C Bus frequency

To use the I2C bus we must first define its frequency of operation.

I2CFreq `( ukHz -- ukHz )`
Set I2C frequency
The desired frequency of operation (in kHz) is popped from the stack. The Gizmo tries to select the maximum available frequency of operation that is below the desired value and selects it for the I2C peripheral. That selected frequency is pushed next on the stack.
The selected is the maximum available frequency that is equal or less that the desired one.
Once a frequency is chosen it is pushed on the stack.

The current implementation of the Gizmo can use four frequency values for the I2C bus. 50kHz and 100kHz for normal speed mode and 200kHz and 400kHz for high speed mode. If you select any other frequency the I2C bus will default to the highest frequency that is equal or less than the desired frequency. If you chose a frequency below 50kHz an error will be shown and the 50kHz clock frequency will be selected.

9.2.2. Address scan

As the number of addresses in the I2C bus if finite, it is suitable to scan all addresses to check the devices on the bus. The ***ISCAN*** command performs a scan on the I2C bus and returns n+1 elements on the stack where n is the number of devices found.

ISCAN `( -- uaddu...uadd1 u )`
Scan for I2C devices issuing a null transfer in each address.
The found I2C addresses are shown on the console.
After the command, the TOS will contain the number ***u*** of valid addresses found on the bus.
The following ***u*** levels of the stack will contain the found addresses.
It is possible for this command to fill the stack so if there are too many devices in the bus you could not get all addresses.

In order to scan the bus the Iscan command performs a dummy transfer to every address from 0x08 to 0x77 with 0 Bytes write and 0 Bytes read. The non correct addresses are detected as in these cases the transfer is not acknowledged by the slave.

Beware that some devices show on the bus as several addresses so you cannot imply that there are the same number of devices than addresses.

9.2.3. I2C Transfers

The SPI bus was symmetric and full duplex. The I2C bus is non symmetric and half duplex. That means that at any time the information can only go in one direction. As the communications are initiated by the master all transfers start with the master sending data. In some cases, when the master stops sending data it switches to read mode and starts receiving data.
The basic I2C transfers send ***uw*** bytes in write mode to the slave and receives ***ur*** bytes in read mode.

I2CTransfer `( uaddr d1w..duw uw ur -- dur...d1r )`
Interchanges data with the slave at ***uaddr*** address. Send ***uw*** bytes ***d1w...duw*** and receive ***ur*** bytes ***d1r..dur***. After the transfer the first received byte is on the TOS.

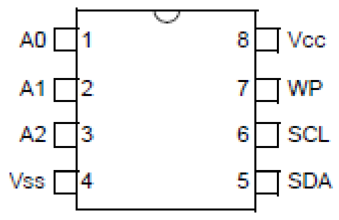
A lot of slave devices use the I2C bus to read or write its internal registers. The following two words are simplified cases of the I2CTransfer word for those special cases.

I2CReadReg `( uaddr ureg -- uval )`
Read register ***ureg*** of the slave with address ***uaddr***. To do that, the ***ureg*** value is sent to the device at ***uaddr*** address. Then the I2C bus changes to read mode and the value ***uval*** is read from the same ***uaddr*** device.

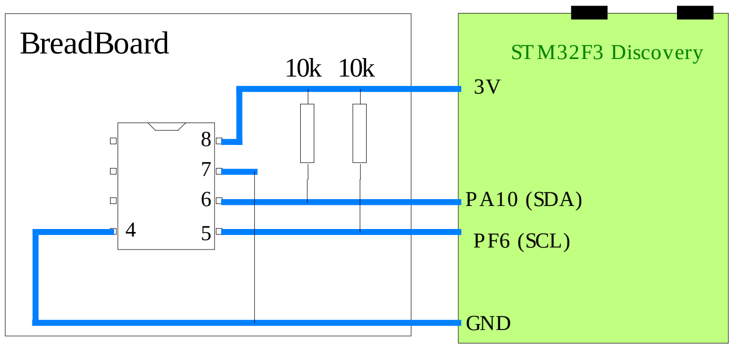
I2CWriteReg `( uaddr ureg uval -- )`
Write value ***uval*** on the register ***ureg*** of the slave at address ***uaddr***. To do that, the ***ureg*** value is sent to the device at ***uaddr*** address followed with the ***uval*** value.

9.2.4. I2C EEPROM Example

To show the operation of the I2C bus we present an example that writes and reads from a Microchip 24LC08B 1kB I2C EEPROM in a DIP8 package.



Pins A0, A1 and A2 are not used in this chip so we can let them alone.
The Write Protect WP pin must be tied to Vss to enable writing to the memory.
The datasheet recommends 10k pullup resistors for 100kHz operation so these are the values we will use.



We set the 100kHz frequency that is compatible with the pull-up resistors:

```
100 I2CFreq
<2> Top: 100
```

The 24LC08B includes 1 kByte of memory addressable in 4 pages of 256 Bytes. Each page uses a different I2C address. The addresses used by this memory have the format:

1 0 1 0 X B1 B0

Where: X : Can be “0” or “1”
B1, B0 : Identify one of the four pages: 00, 01, 10 or 11

That means that the device occupies the following eight addresses:

Page	**Addresses**
0	0x50 and 0x54
1	0x51 and 0x55
2	0x52 and 0x56
3	0x53 and 0x57

We can use the ***ISCAN*** word to check the available addresses:

```
>ISCAN
Addr Found: 0x50
Addr Found: 0x51
Addr Found: 0x52
Addr Found: 0x53
Addr Found: 0x54
Addr Found: 0x55
Addr Found: 0x56
Addr Found: 0x57
<9> Top: 8

>.S

<9> 80 81 82 83 84 85 86 87 8
<9> Top: 8
```

As we can see the ***ISCAN*** word detects all eight memory addresses.

To write in a memory position we must send the address followed by the Byte to write.
The memory addresses span from 0x000 to 0x3FF. The first digit is the page number and the two last ones are the position within this page.

To write, for instance 0xAA at the address 0x172 we must write at address 0x72 of page 0x1

```
0x51 0x72 0xAA    \Address   Page Pos  and   Value
2 0 I2CTransfer   \Transfer command
```

To read at the same address we must send the same address and page position and change to read mode to get the value at the selected memory position.

```
0x51 0x72         \Address  and  Page Pos
1 1 I2CTransfer   \Transfer command
X.                \Write TOS in hexadecimal

AA <Empty Stack>
```

We get the same value we have written before. As this is an EEPROM memory we get the same value if we turn off the power, reboot and read again.

10. Board Peripherals

The STM32F3 Discovery board features several peripheral devices around the main MCU chip. Those include a Gyroscope and a combined Accelerometer / Magnetometer. The F3 Gizmo can access and read them.

10.1. Gyroscope

This section will deal with the L3GD20 Gyroscope that is included in the board. This device can measure 3 axis angular rate of movement with 16 bit resolution and with selectable full scales of 260, 500 and 2000 dps (degrees per second).

The device is connected to the MCU through the SPI#1 channel. This channel is independent of the SPI#3 used in the above described SPI commands so you cannot access this peripheral using the SPI... commands.

The Gizmo configures the Gyroscope on start-up to work on the 260 dps scale so in the sensor reading one LSB corresponds to 8,75 mdps (millidegrees per second).

10.1.1. Basic Gyroscope reading

The ***GyroRead*** command reads the gyroscope angular rates in the three axes.

GyroRead `( -- nz ny nx )`
Reads the gyroscope in the three axes.

The Gyroscope has some offset error that makes it return a turn rate when no rotation is present at all. The ***GyroZero*** word can be used to set the current readings as the zero eliminating the offset error.

GyroZero
Sets gyroscope zero to current read value.
The normal use of this command is to null the zero offset of the sensor when we know that the gyroscope is not moving.
Setting the zero using this word affects any posterior use of the GyroRead command.

GyroSetZero `( nx ny nz -- )`
Manually set the zero of the Gyroscope for the three axes.

Remember that the ***GyroRead*** word includes the zero offset correction so in order to manually adjust the zero using the ***GyroSetZero*** it is best to eliminate the zero compensation before reading the gyroscope.

```
0 0 0 GyroSetZero       \Eliminate zero compensation
GyroRead { z0 y0 x0 }   \Read Gyroscope and set three locals
x0 y0 z0  GyroSetZero   \Set the zero compensation
```

10.1.2. Access to gyroscope internal registers

The gyroscope words included in the F3 Gizmo only uses a little part of the available functionalities of the L3GD20 device. The following two words give full access to the gyroscope internal registers.

Those words threat all registers as unsigned 8 bit registers so they return and set values from 0 to 255.

GyroReadReg `( ureg -- uval )`
Reads the gyroscope internal register ***ureg***.

GyroWriteReg `( ureg uval -- )`
Writes the gyroscope internal register ***ureg*** with the ***uval*** value.
It only enables writes to correct registers as writing to a reserved or non writable register can damage the device.

10.1.3. Gyroscope Examples

The ***gdump*** word writes on the console the Gyroscope readings two times per second.

```
\ Continuous read each 0.5s
: gdump
BEGIN
  GyroRead
  ."X: " . ."Y: " . ."Z: " . CR
  500 MS
AGAIN ;
```

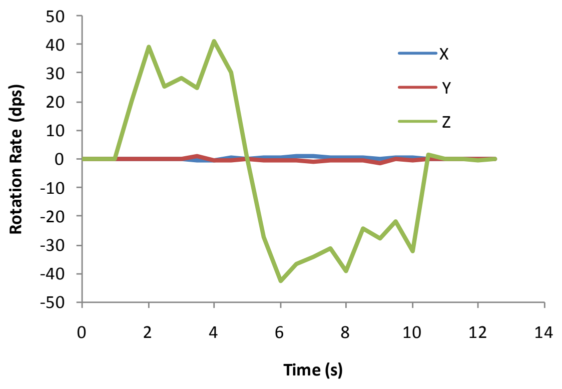
If you rotate the board you will get a high value reading in the axes you are rotating arround.

We can set the zero before executing the word to set the zero value:

```
GyroZero
gdump
```

You will see that the offset has been greatly reduced.

In the following figure we have rotated the board around the Z axis (orthogonal to the board plane) first clockwise then anticlockwise. You can see that only the Z turn rate has been detected. In the figure a factor 0,00875 has been applied to all read data in order to show the values in degrees per second (DPS).



To test the direct reading of the internal Gyroscope registers we want to read the WHO_AM_I register at position 0x0F. This register should read 0xD4.

```
\Read Who Im I
0x0F GyroReadReg X.
D4 <Empty Stack>
```

We can also read the register that measures the internal device temperature in ºC at position 0x26.

```
\Read temperature
0x26 GyroReadReg .
24 <Empty Stack>
```

We can disable the Gyroscope writing 0x00 to the CTRL_REG1 at position 0x20:

```
0x20 0x00 GWR

gdumpc
X , Y , Z
-28  , -19  , 20
-28  , -19  , 20
-28  , -19  , 20
-28  , -19  , 20
-28  , -19  , 20
```

The Gyroscope gives now only the zero rate values and is not sensible to any rotation.

10.2. Accelerometer

The STM32F3 Discovery includes a LSM303DLHC device that includes an accelerometer and a magnetometer. This device connects to the STM32F303 MCU using the I2C1 peripheral. This I2C bus is independent to the I2C2 peripheral we described in the above I2C section so you cannot use the I2C... words with this device.
Although the accelerometer/magnetometer is only one chip, it uses two I2C addresses, one for the accelerometer and one for the magnetometer.

By default the accelerometer has an end of scale is +/-2g. Full scale is 32768 for +2g so that gives 61 ug/LSB.

10.2.1. Basic Accelerometer reading

The words to access the accelerometer are very similar to the ones on the gyroscope.

ACcelRead `( -- nz ny nx )`
Reads the accelerometer three axes.

ACcelZero
Set as zero the current X and Y readings.
The normal use of this word is calling it when the board is horizontal and stable.
The Z axis is no zeroed as it should be 1g and not zero at horizontal position.

ACcelSetZero `( nx ny nz -- )`
Manually sets the zero reading for all three axes.

Remember that, as in the case of the gyroscope, the ***ACcelRead*** word includes the zero offset correction so in order to manually adjust the zero using the ***ACcelSetZero*** it is best to eliminate the zero compensation before reading the accelerometer.

Observe that the first C is in upper case, so the contracted version for these words are: ***ACR***, ***ACZ*** and ***ACSZ***.

The following code sets the zero to all three axes so that it reads 0 in x and y and 1g (16384 reading) in the z axis. We use the short names for the Accelerometer words. This code is different from the one in ACcelZero because it also compensates the Z axis offset.

```
: ACompensate
0 0 0 ACSZ                \Eliminate zero compensation
ACR { z0 y0 x0 }          \Read Accelerometer and set three locals
x0 y0 z0 16384 - ACSZ ;   \Set the zero compensation

ACompensate
CLEAR ACR .S CR           \Show current read
```

10.2.2. Access to accelerometer internal registers

The accelerometer words included in the F3 Gizmo only uses a little part of the available functionalities of the LSM303DLHC device. The following two words give full access to the accelerometer internal registers.

The two following words threat all registers as unsigned 8 bit registers so they return and set values from 0 to 255.

ACcelReadReg `( ureg -- uval )`
Reads the accelerometer register ***ureg***

ACcelWriteReg `( ureg uval -- )`
Write the accelerometer register ***ureg*** with the ***uval*** value
It only enables writes to correct writable registers.

10.2.3. Accelerometer Examples

The accelerometer measures any acceleration in the three axes. That includes the gravity acceleration. The 1g gravity acceleration decomposes in the three axes depending on the orientation of the board. If the board is parallel to the floor Z will measure 1g and the X and Y axes will measure zero.

We can use a similar word than in the Gyroscope to show the acceleration each half second.

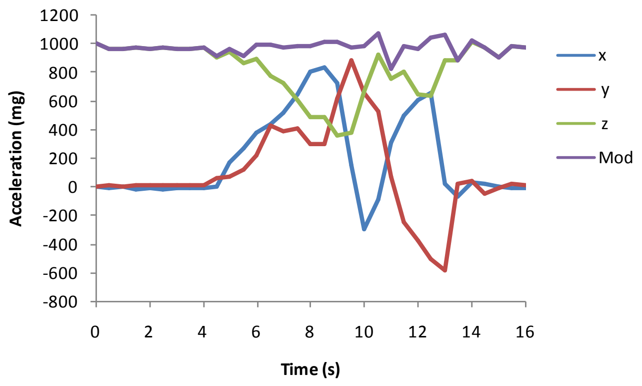
```
\ Continuous read each 0.5s
: adump
BEGIN
  AccelRead
  ."X: " . ."Y: " . ."Z: " . CR
  500 MS
AGAIN ;

adump
X: -640 Y: -320 Z: 14848
X: -448 Y: 0 Z: 15232
X: -512 Y: 0 Z: 15168
X: -384 Y: 64 Z: 15232
X: -384 Y: 0 Z: 15232
X: -384 Y: 64 Z: 15296
X: -640 Y: 192 Z: 15104
X: -320 Y: -320 Z: 15296
```

With the board parallel to the floor we observe that there is some error in the X and Y axes and that the Z axis is the bigger one.
We can develop a program that calibrates the accelerometer and then measures the acceleration in mg units (1/1000 g). Inside the new ***adumpCal*** word we will use the previously described ***Acompensate*** word instead of the standard ***ACcelZero***.

```
\ Continuous read each 0.5s
: adumpCal
ACompensate
."x , y , z " CR
BEGIN
  AccelRead
  1000 * 16384 / . ." , "
  1000 * 16384 / . ." , "
  1000 * 16384 / . CR
  500 MS
AGAIN ;
```

Executing this program, moving the board and showing the data using a spreadsheet we can compose the following graph:

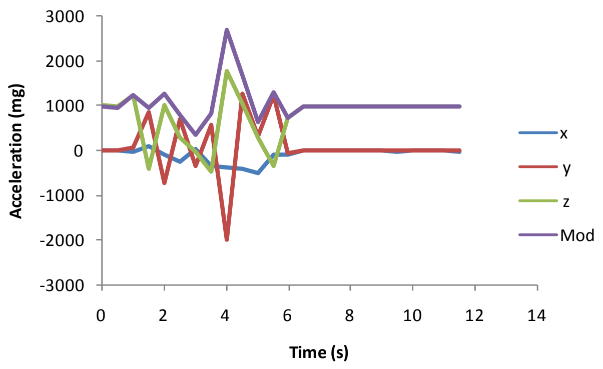


In the graph the three x, y and z accelerations are shown together with the modulo defined as:

$$Mod = \sqrt{A_X^2 + A_y^2 + A_z^2}$$

As we can see, as we are moving the board slowly, the only gravity it senses is the gravitational force and this is always 1g although decomposed in the three axes depending on the board orientation. From this data the inclination angles of the board can be calculated but that goes beyond the objectives of this example.

If we moved the board with force we can find a graphic like the following one.



Here we observe that the modulo goes beyond and below 1g. That means that there is some acceleration into play that is not the earth gravity.

10.3. Magnetometer

As we have explained the LSM303DLHC device includes a magnetometer next to the described accelerometer.

10.3.1. Basic Magnetometer reading

The F3 Gizmo configures the magnetometer with 75Hz update rate and a +/- 2.5 gauss range. That gives 670 LSB/gauss in the X and Y axes and 600 LSB/gauss in the Z axis.

MagRead `( -- nz ny nx )`
Reads the magnetometer's three axes.

MagSetZero `( nx ny nz -- )`
Manually set the magnetometer zero value.
It will affect any posterior reading using MagRead.

10.3.2. Access to accelerometer internal registers

The magnetometer words included in the F3 Gizmo, as in the case of the gyroscope and the accelerometer, only uses a little part of the available functionalities of the LSM303DLHC device. The following two words give full access to the magnetometer internal registers.

The two following words threat all registers as unsigned 8 bit registers so they return and set values from 0 to 255.

MagReadReg `( ureg -- uval )`
Reads the magnetometer register ***ureg***

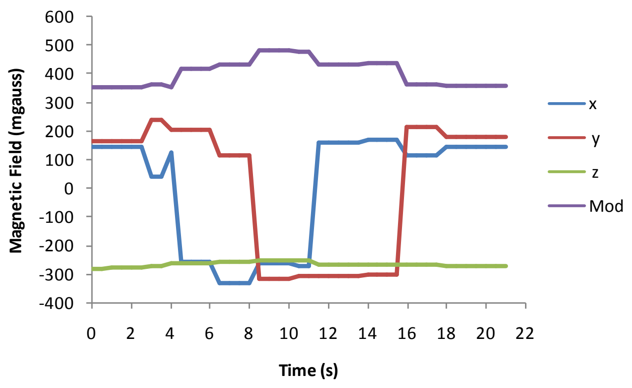
MagWriteReg `( ureg uval -- )`
Write the magnetometer register ***ureg*** with an ***uval*** value
It only enables writes to correct writable registers

10.3.3. Magnetometer Example

We can develop a ***mdump*** word similar to the previous cases to show the evolution of the magnetic field with time. We will design the word so that it gives the output values in mgauss units (1/1000 gauss).

```
\ Continuous read each 0.5s
\ Gives values in mGauss
: mdump
."x , y , z " CR
BEGIN
  MagRead
  1000 * 670 / . ." , "
  1000 * 670 / . ." , "
  1000 * 600 / . CR
  500 MS
AGAIN ;
```

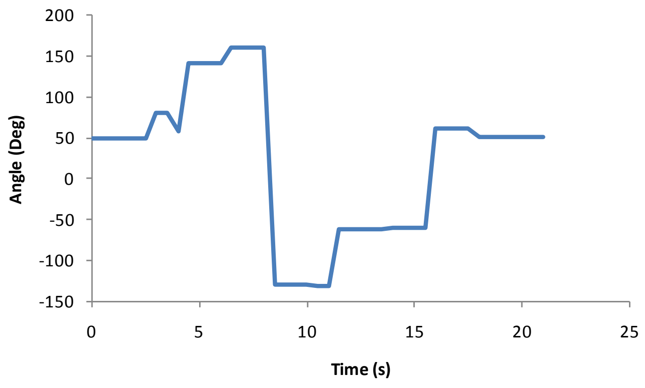
As you can find in the Wikipedia the earth's magnetic field goes from 250 to 650 mgauss. The follow figure shows the measured magnetic field in the three axes together with the modulo of the three components. During the experiment the board has been rotated around the z axis in 4 90º steps to complete a full rotation.



As we can see the rotations have been performed at 2s, 8s, 11s and 16s.
As we have the Z axis has not changed it always measures the same. The modulo changes a little but not much.

We can compute the angle or the projection of the magnetic field on the X-Y board plane.

$$Angle = atan\frac{x}{y}$$



We can see that the angle in a full rotation is monotonic. It goes from 50º to 150º, then goes to -140º , -50º and finally the start position of 50º. As expected each rotation is about 90º to give a full complete 360º rotation.

11. Thread synchronization

The use of threads enables us to run independent works in the microcontroller easing its programming. It can however give its own problems. **Semaphores** and **mutexes** ease to solve some of these problems.

11.1. Binary Semaphores

One of the problems that a multithreaded program must solve is thread communication. Sometimes one thread must indicate some information to another or must wait until another thread has done some work.

Binary semaphores provide basic communication between threads.
The Gizmo includes NSEM binary semaphores. The first one has number 0 and the last one has number NSEM-1. To know the exact number of semaphores that have been compiled in the current Gizmo version you can use the ***LIMITS*** word.

Each semaphore can be in two states, **FREE** or **TAKEN**. To know the status of each semaphore you can use the ***SLIST*** word.

SLIST
List status for all semaphores.
This word cannot be used inside a word definition.

There are two basic words to use a semaphore: ***SIGNAL*** and ***WAIT***.

SIGNAL `( us -- )`
Signals the semaphore ***us*** making it **FREE**.

WAIT `( us -- )`
Wait for semaphore ***us*** making it **TAKEN**.
If the semaphore was **FREE** before calling this word, it sets it **TAKEN** before returning.
If the semaphore was **TAKEN** before calling this word, the word (and the associated thread) blocks waiting for the semaphore to be **FREE** before taking it.

Sometimes it is important to be able to reset all semaphores to the FREE status. That can be done using the ***SRESET*** word.

SRESET
Reset all semaphores to **FREE** status.

Semaphore example:

We can create a ***S1*** word that waits for semaphore 0 and then lights the LED 0 during half a second after repeating the sequence. We can call this word in a background thread:

```
\ Wait for S0 and light led 0 500ms
: S1 BEGIN 0 WAIT 0 LS 500 MS 0 LC AGAIN ;
THREAD T1
```

The thread will light the LED only once as after the first WAIT the semaphore will be taken and that will block the thread.
To unlock the thread we can signal the semaphore. For each signal we call the LED will blink once.

```
\ Signal to start LED
0 SIGNAL
```

11.2. Mutexes

All threads can access to all MCU resources and some of them cannot be accessed at the same time without running into problems. For instance you cannot start an I2C transfer from one thread when an I2C transfer is ongoing on another. The regions of the code that cannot be run at the same time by different threads are **critical** code zones.

To protect critical zones we use **mutexes**.
The Gizmo includes NMTX mutexes. The first one has number 0 and the last one has number NMTX-1. To know the exact number of mutexes that have been compiled in the current Gizmo version you can use the ***LIMITS*** word.
Different operating systems feature different kinds of mutexes. The mutexes in the F3 Gizmo are provides by the ChibiOS/RT operating system so you can read this operating system documentation to learn more about them.

A mutex is somewhat similar to a semaphore. But the nomenclature is a little different. Instead of taking a mutex you **lock** it and instead of signaling a mutex you **unlock** it.

Let's say that you have two treads that need to access the I2C bus but not at the same time. We select for instance the mutex number 0 to control the access to the I2C bus. To be sure that only one thread access the bus at the same time each thread must **lock** the mutex before accessing the bus. If the mutex was **unlocked** when the thread tries to lock it the mutex is locked and the thread can use the bus. If the mutex was **locked** when the thread tries to lock the mutex the mutex will block the tread and it will be suspended until the mutex is available to be locked.

Mutexes are different to semaphores because they are owned by the thread that locks them. That means that a mutex can only be unlocked by the thread that locked it. Moreover, to prevent some kind of nasty problems the mutexes must be unlocked in stack of LIFO fashion. That is, you must unlock the last mutex you locked.

There are two words to **lock** and **unlock** mutexes:

LOCK `( u -- )`
Lock mutex ***u***. If the mutex was unlocked it will be locked and owned by the calling thread with will continue running. If the mutex was locked the calling thread will blocked until the mutex is unlocked and can be locked by the thread.

To prevent deadlocks, if two treads have to lock at the same time two or more mutexes they have to lock them in the **SAME** order. Think about what could go wrong if you don't lock mutexes that way. You have been warned.

UNLOCK
Unlock last owned locked mutex. You don't need to specify the mutex number as you always unlock the last locked mutex.

Sometimes you want to unlock all owned mutexes. To do that you can use the ***UNLOCKALL*** word. When one thread ends all owned mutexes are unlocked.

UNLOCKALL
Unlock all owned mutexes. This word is automatically called on thread end.

Mutexes also feature priority inheritance to prevent priority inversion hazards. An example will explain what it is all about: Let's say that you have a low priority thread, a high priority thread and a mutex. The low priority thread locks the mutex, then the high priority thread tries to lock the mutex and, as is already locked, it blocks. If now a medium priority thread runs, it will lock the low priority thread and that will prevent it to unlock the mutex that the high priority thread is waiting for. There is priority inversion because the medium priority thread runs and the high priority thread is blocked. The problem is that you don't know when the low priority thread will be able to run to unlock the thread and let the high priority thread run.

To prevent this problem the mutexes implement the priority inheritance. When a low priority thread locks a mutex and another higher priority thread tries to lock this mutex the low priority inherits the priority of the higher priority thread to until the mutex is unlocked. That way the priority inversion is prevented and the high priority thread is able to run as soon as possible.

Mutexes example:

Mutexes are used to prevent resource collisions. The bad thing about mutexes is that if you don't use mutexes the program usually works OK for a time. Sometimes for a long time. That means that a resource collision problem can pass through all the program debug process.
To test the use of mutex we must make collisions more frequent.

The following code draws two squares on the screen, one filled with “O”s and one filled with “X”s using two threads. To do that, two words are implemented: ***osquare*** and ***xsquare***. The word ***osquare*** runs in a background thread and the ***xsquare*** word runs on the main thread.

```
: osquare
255 Verbose
BEGIN
  10 2 DO
     2 i AT-XY
     20 2 DO
       ."O"
       1 MS
     LOOP
  LOOP
AGAIN ;

: xsquare
255 Verbose
BEGIN
  10 2 DO
     25 i AT-XY
     45 25 DO
       ."X"
       1 MS
     LOOP
  LOOP
AGAIN ;

PAGE THREAD osquare xsquare
```

As both threads access the screen they interact badly and you won’t see two different squares. To stop the programs just push the USER button.

To prevent mutual interactions between in the critical zone between the AT-XY command and the end of the inner loop mutexes can be used.

The following code implements two new words ***osquare2*** and ***xsquare2*** that prevents bad interaction by the use of mutexes.

```
: osquare2
255 Verbose
BEGIN
  10 2 DO
     0 LOCK     \We get exclusive screen access here
     2 i AT-XY
     20 2 DO
       ."O"
      \ 1 MS
     LOOP       \We leave exclusive screen access here
     UNLOCK
  LOOP
AGAIN ;

: xsquare2
255 Verbose
BEGIN
  10 2 DO
     0 LOCK       \We get exclusive screen access here
     25 i AT-XY
     45 25 DO
       ."X"
      \ 1 MS
     LOOP
     UNLOCK      \We leave exclusive screen access here
  LOOP
AGAIN ;

PAGE THREAD osquare2 xsquare2
```

If you run this program you should see that it behaves as expected

12. REFERENCES

This last section gives some references related to the F3 Gizmo.
The CForth engine references are included in the independent CForth manual.

The provided links work at the time of creating this document but nothing on Internet is eternal. If any link don't work anymore just use any search engine.

ChibiOS/RT

ChibiOS/RT is a Real Time Operating System (RTOS). It provides OS services as a scheduler, threads, semaphores, mutexes and so on. It also provides the supported microcontrollers with a Hardware Abstraction Layer (HAL).
This RTOS supports most of the STM32 Discovery boards so it is a good foundation to any project based on these boards.
You can find all the required information about ChibiOS/RT in its web page:
http://www.chibios.org

STM32F3 Discovery Board

The STM32F3 Discovery board is a very low cost high performance evaluation board.
The F3 Gizmo is a firmware only solution for this board. All the credit for this fantastic board hardware goes to ST.
You can find information about this board in the ST Webpage:
http://www.st.com/web/en/catalog/tools/FM116/SC959/SS1532/LN1848/PF254044

STM32F303 Documentation

The STM32F3 Discovery board is built around the STM32F303VCT6 microcontroller (MCU). You don't need the MCU documentation to use the F3 Gizmo but it can help. If you plan to do some serious work on this board, however, you are encouraged to read its documentation.
You can find information about this MCU in the following link:
http://www.st.com/web/catalog/mmc/FM141/SC1169/SS1576/LN1531/PF252054
As in all ST MCUs the documentation is divided in two important documents:

Reference Manual (RM0316)
This document includes generic information for a family of MCUs.
DM00043574.pdf

STM32F303 Datasheet
This document includes all specific information for a particular MCU within a family.
DM00058181.pdf

STM32F3 Discovery Peripherals

The STM32F3 Discovery board includes two main peripherals aside from LEDs and buttons.

The L3GD20 Gyroscope is a three axis digital output gyroscope that can be accessed by SPI or I2C serial buses.
You can find more information about it in its datasheet.

The LSM303DLHC is a combined digital output 3D Accelerometer and Magnetometer that can be accessed using an I2C serial bus.
You can find more information about it in its datasheet.

If you want to get all these devices can give you should access them using their internal registers so you will need to read their datasheets.

Serial Memories

In the SPI and I2C serial buses we used two Microchip memories. We provide here their references and datasheet links:
SPI Memory

23k640
This is a SPI 8k Byte RAM memory. We use it in the DIP8 package option.
You can find more information in its datasheet.

24LC08B
This is an I2C 1k Byte EEPROM. We use it in the DIP8 package option.
You can find more information in its datasheet.