KDE485 Reference Manual

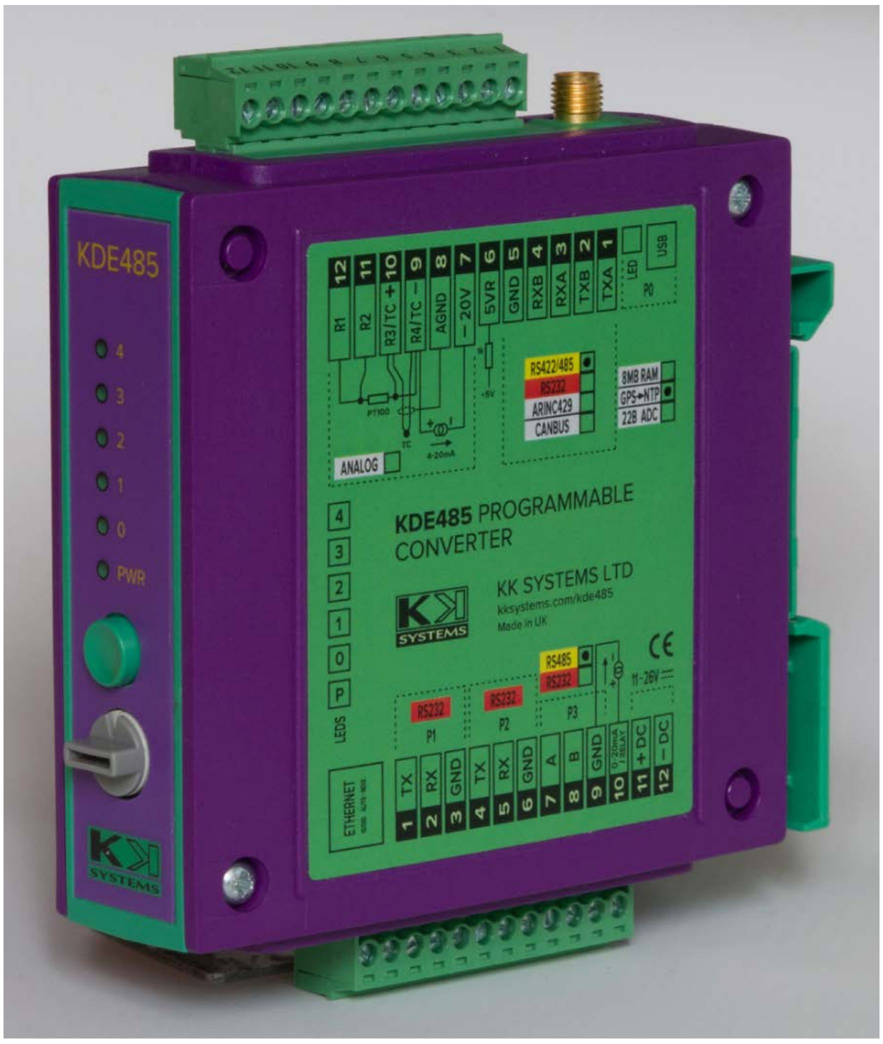
1st February 2026

https://kksystems.com/kde485

Table of Contents

KDE485 Overview

To get started quickly: review pages 1-25



Hardware

The KDE485-PROG is a C user-programmable platform, with the following interfaces:

Four serial ports (2xRS232, 1x2wRS485, 1x4wRS485/RS422 - optionally 4xRS232)
Ethernet (10/100, auto-MDIX)
Micro-USB (2MB FLASH drive and a serial VCP implemented)
Analog sensor interface, 16-bit isolated
PT100, PT1000, thermocouple (all 8 types), AD590, 4-20mA, 2 voltage inputs
Analog current sink 0-20mA for 4-20mA sensor emulation / relay drive
Various internal expansion options: 2x 12-bit ADC, 2x 12-bit DAC, any SPI device, CAN
Five addressable LEDs, a push button switch, a 16-position hex switch
ARINC429 (factory option)
8MB SPI RAM (factory option)
SPI GPS (factory option; uses external GPS antenna - shown in above photo)

Power input is 11-35V DC, isolated
-20V output is provided for powering 4-20mA sensors (requires 24V DC power)

Processor

STM 32F417 running internally at 168MHz. This is a very fast ARM32 32-bit processor which does most operations (including a 32 bit integer or single precision floating point multiply) in 1 clock cycle. Features used in the KDE485 include:

- 1MB FLASH for code of which around 50% is available for user applications
- 256k RAM
- Ethernet 10/100
- USB 2.0 FS
- four UARTs
- SPI controllers (up to 21MHz)
- RTC with a "supercap" backup
- two 12 bit ADCs and two 12 bit DACs (internal expansion)
- DMA controllers, timers, etc
- hardware crypto acceleration
- SWD single wire debugging (internal connector)
- CANbus and I^{2}C
- programmable hardware watchdog

Development Environment

The "development kit" is based on the STM32 Cube IDE (integrated development environment).
https://www.st.com/en/development-tools/stm32cubeide.html
Cube IDE v1.14.1 is used.
It uses the ARM32 GCC compiler (v11) with integers up to uint64_t and single and double floats.
It is completely free and license-unrestricted.

User software is compiled within Cube IDE to a binary file which is transferred to the target by placing it into a 2MB FLASH filespace and activating an update sequence which flashes the CPU from this file. The filespace can be accessed via USB or by accessing an HTTP server with a browser. The original factory software remains stored separately in a reserved FLASH area and can always be restored even if user software has "locked up" the unit.

For long term development environments, and where the debugging features of CubeIDE are required, the SWD debugging interface (similar to JTAG) is available on an internal 10-pin connector compatible with the ST-LINK V2 or V3 GDB debugger.

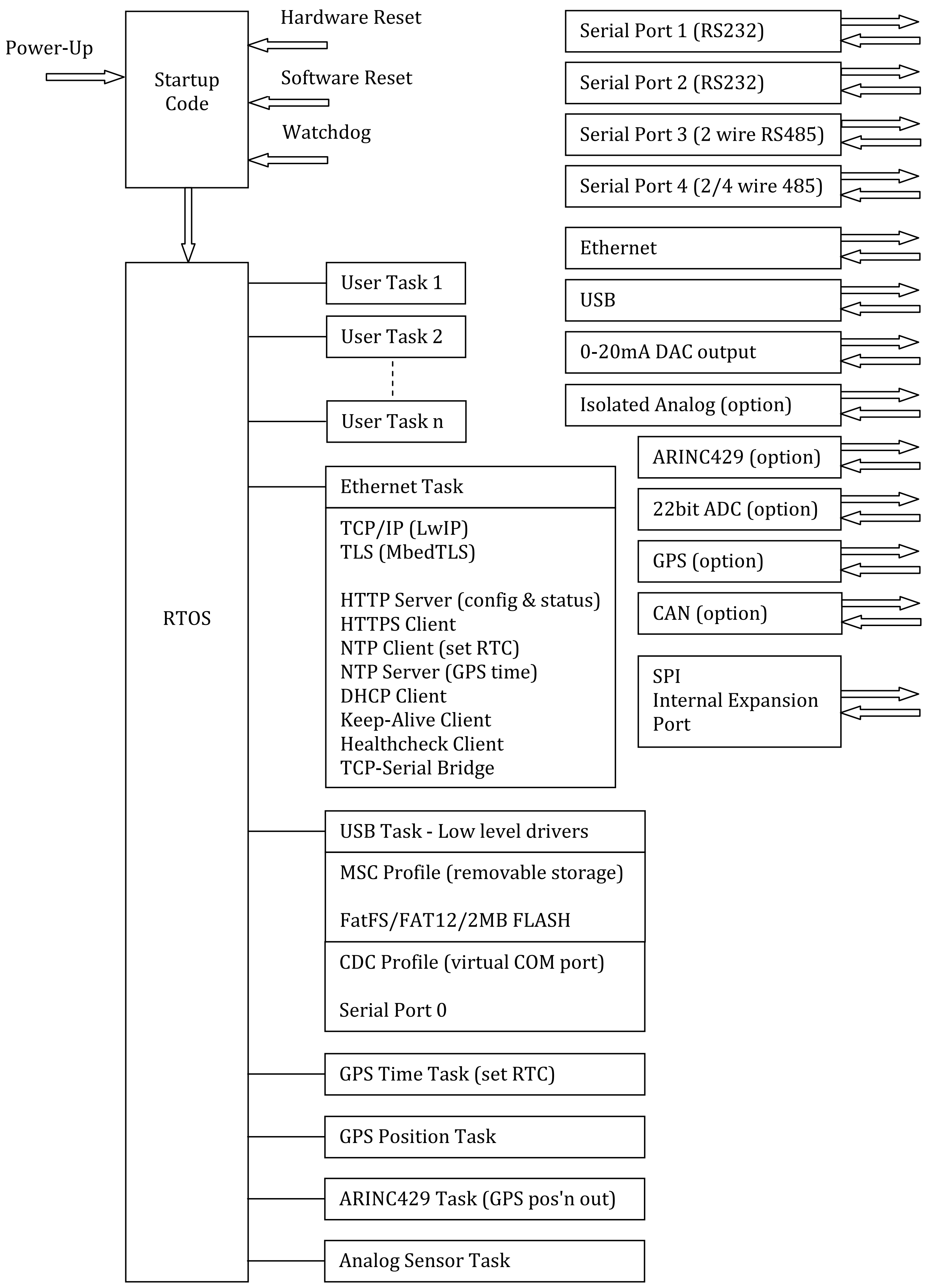
Much of the software in the KDE485 comprises of open-source modules proven over many years, in millions of commercial and industrial products. These modules are published in source form and all those sources are provided within the KDE485 development kit.

Thread or Task?

Throughout this manual the words Thread and Task are used interchangeably. Both refer to RTOS processes. The expression "thread-safe" means the code is re-entrant and can be freely called by multiple RTOS tasks. This can be achieved in various ways, including mutexes.

Software Architecture

User applications run under a real time operating system: FreeRTOS. It is easy to create applications and each one can be written as if it "owns" the computer. The transition from writing traditional software (the "main loop" or a state machine) is straightforward.


Power-Up
Startup Code
Hardware Reset
Software Reset
Watchdog
RTOS
User Task 1
User Task 2
User Task n
Ethernet Task
TCP/IP (LwIP)
TLS (MbedTLS)
HTTP Server (config & status)
HTTPS Client
NTP Client (set RTC)
NTP Server (GPS time)
DHCP Client
Keep-Alive Client
Healthcheck Client
TCP-Serial Bridge
USB Task - Low level drivers
MSC Profile (removable storage)
FatFS/FAT12/2MB FLASH
CDC Profile (virtual COM port)
Serial Port 0
GPS Time Task (set RTC)
GPS Position Task
ARINC429 Task (GPS pos'n out)
Analog Sensor Task
Serial Port 1 (RS232)
Serial Port 2 (RS232)
Serial Port 3 (2 wire RS485)
Serial Port 4 (2/4 wire 485)
Ethernet
USB
0-20mA DAC output
Isolated Analog (option)
ARINC429 (option)
22bit ADC (option)
GPS (option)
CAN (option)
SPI
Internal Expansion
Port

API

The KDE485 API supports all of the above mentioned hardware features, including

- Ethernet (LwIP, TLS)
- FLASH drive FAT file system
- Linear FLASH data logging area
- Debug output via USB VCP
- ARINC429 support
- NMEA U-BLOX and SPI GPS support
- Data upload to Dropbox
- Setting up an HTTPS Client process

No direct hardware access is needed to create applications, but it can be used if desired. All 32F417 features are available, documented in its Reference Manual - **RM0090**. Various C source code examples are included.

Factory Software Updates

Factory software can be updated using the same method as for user software. It is a different filename and there is some more security around it. One cannot generate the factory software using the distributed Cube IDE kit.

U**ser Program Security**

Anyone can purchase a KDE485 but no function is provided for the extraction of a user application from a programmed KDE485.

Safety

The KDE485 is not authorised for use in any application where injury or death could result from its failure. None of its interfaces should be connected to a hazardous potential.

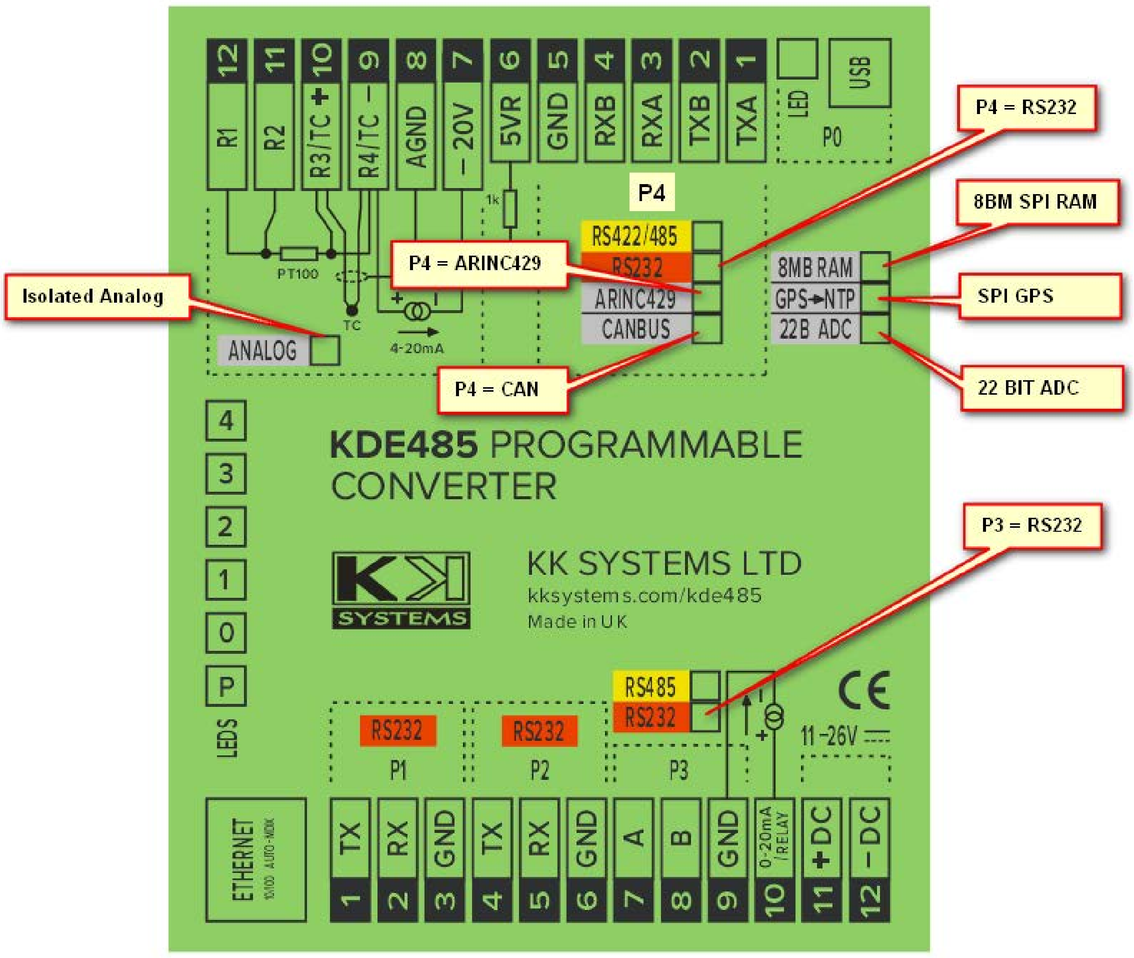
Factory Hardware Options

Several factory installed hardware options are available. These are shown with blobs on the KDE485 side label. The following example label shows:

- Isolated Analog Subsystem
- Port 3 = RS232 (normally RS485)
- Various Port 4 (normally RS422) options: RS232, ARINC429, CAN
- 8MB SPI RAM
- SPI-GPS (normally used for the GPS-NTP version of KDE485)
- 22 bit ADC (replaces SPI-GPS on the SMA connector)

In addition to the above, a reference design and an API is available for an SPI-connected daughter board which is similar to the KDE485's optional analog subsystem and contains:

6 digit 7 segment display (STLED316)
PT100/AD590/ Thermocouple/voltage (16-bit ADC - ADS1118)
22-bit ADC (MCP3550 with ADR441 precision reference)



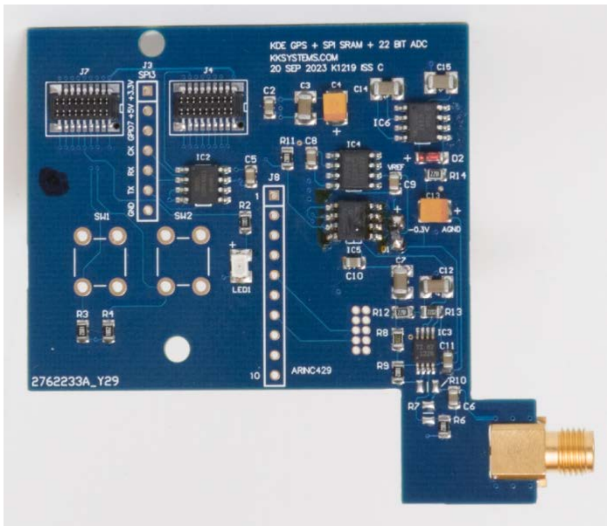
The KDE485 can also be built with a 32F437 CPU which has extra 64k RAM.

Ordering Information and Examples

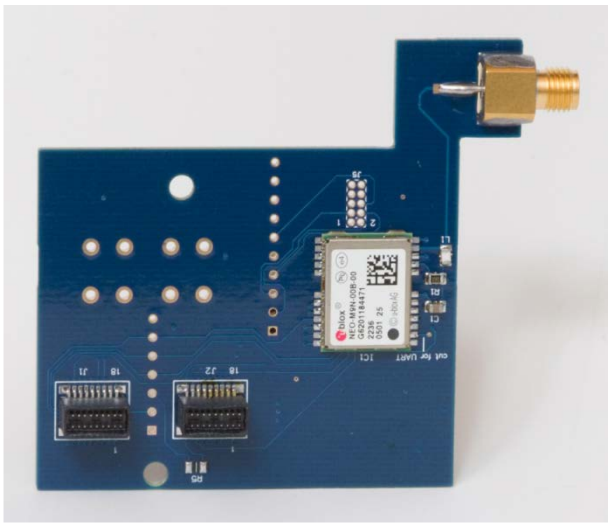
KDE485	Standard product with no options
KDE485-AN	Adds isolated analog option
KDE485-429	Adds ARINC429 option
KDE485-8MB	Adds 8MB SPI RAM option
KDE485-GPS-NTP	GPS to NTP time server version

Add-on Option Boards (normally factory installed)

22 bit ADC



GPS



Custom Products

We always welcome custom product enquiries. Since starting in 1991 we have delivered a large number of these, in production volumes.

The hardware of the KDE485 isparticularly suitable for all kinds of custom products, with user interfaces including keypads and LCD displays.

Specification

Serial Ports

Four asynchronous ports: RS232, RS232, 2 wire RS485, 4 wire RS422/485. Hardware handshake available on port 2. See **Serial Ports** section. Standard interface chips used with +/-8V output on RS232 and 0-5V on RS4xx (MAX232, MAX3089 or equivalent).

Ethernet

10 base T / 100 base T, auto-MDIX

USB

Micro-USB, slave device. CDC (virtual COM port) and MSC (removable storage device).
USB power can be used for the KDE485 (except isolated analog functions) from a 500mA capable USB host port (typ. current 70mA).

0-20mA Current Sink

MOSFET, sinks current into digital ground. Maximum supply voltage to load +30V.
Accuracy 0.05% typical at +20°C, 0.5% over temperature.

Relay Drive

Maximum supply voltage to coil +30V. Maximum load current 100mA.

Isolated Analog Sensor Interface

PT100: accuracy 0.1°C typical at +20°C, 0.3°C over temperature
PT1000: accuracy 0.3°C typical at +20°C, 1°C over temperature
Thermocouple: accuracy 1°C typical at 20°C for J/K types
AD590: accuracy 0.5°C typical at 20°C, + error in AD590 and reference resistor
4-20mA: accuracy 0.03% typical at 20°C, 0.1% over temperature
Voltage inputs: accuracy 0.02% typical at 20°C, 0.1% over temperature

Isolation

64V PK, tested at > 1000V AC RMS, 1 second. 2500V AC RMS by design.

Environmental

Operating temperature -25C to +50C. Storage temperature -40C to +70C.
Relative humidity (operating and storage) 0 to 90% non-condensing.

Ventilation

Rail-mounted KDE485 must have a 50mm gap above and below

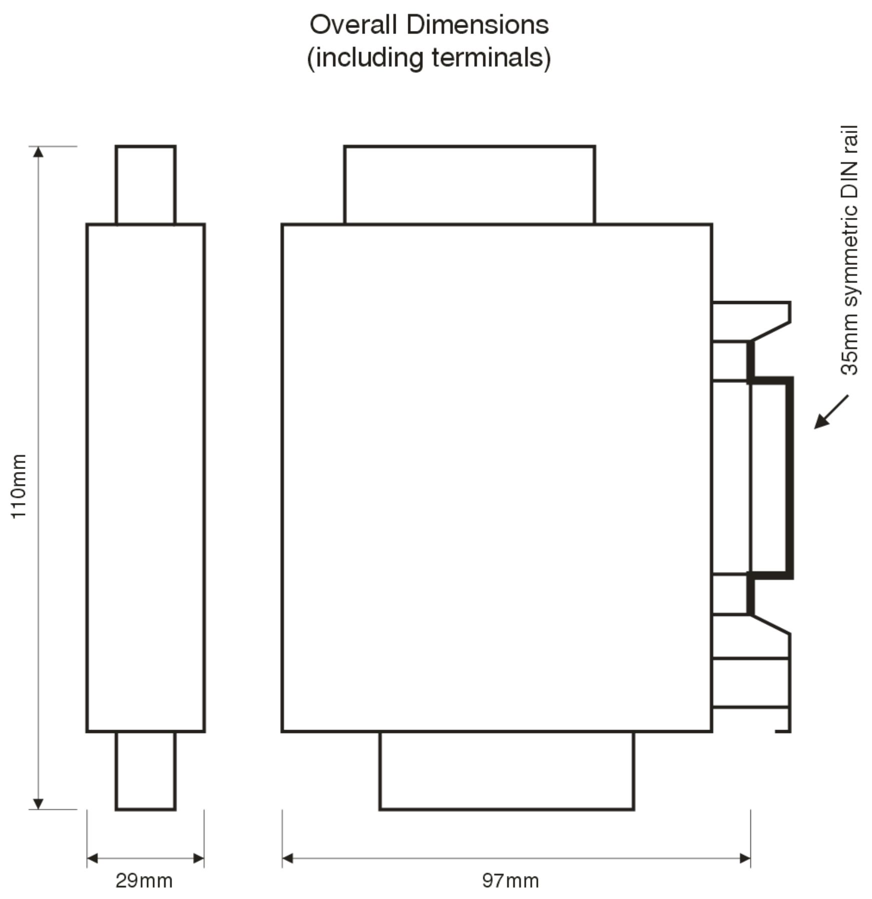
CE Compliance

Emissions EN61000-6-4:2007 Immunity EN61000-4-2:2010
EMC Directive CE2014/30/EU ROHS/REACH 2011/65/EU

Power Input

11-35V DC. Power required 2W.
Recommended voltage 12 or 24V. 24V required for -20V output (isolated analog option).
Startup current 600mA. Input voltage must reach 10V within 1 sec for power supply startup.

Dimensions



Differences between KDE485 and KD485-PROG

The KD485-PROG is a user-programmable protocol converter from KK Systems Ltd which has been in production since 1997 and remains in production. KD485-PROG programs can be ported easily to the KDE485. The main differences are

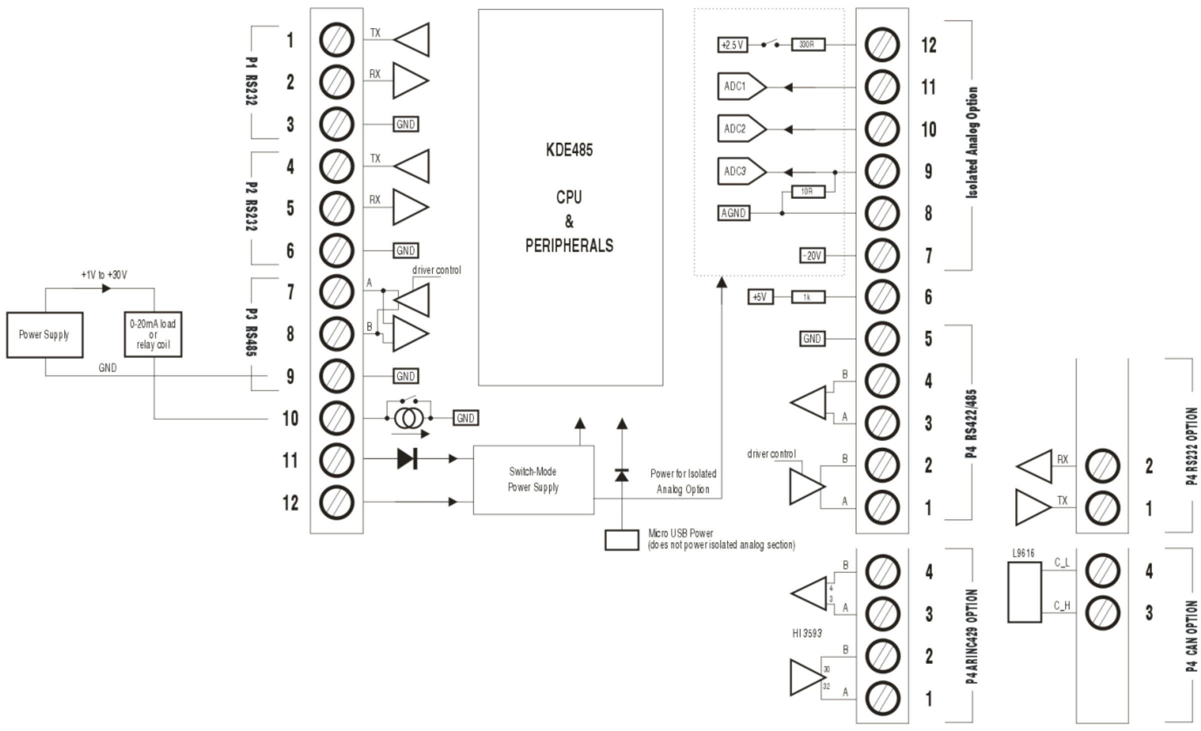
- KDE485 programs run 10x to 100x faster
- much more code and data space
- with the RTOS, writing complicated code is much easier
- a powerful graphical environment for writing and debugging (batch file on KD485-PROG)
- four serial ports (KD485-PROG has two) plus USB Virtual COM Port (CDC)
- baud rates below 1200 baud, and xon/xoff, are not available on the KDE485
- 20mA (TTY loop) option not available on KDE485
- slight differences in some of the API calls (KDE485 has auto driver control on RS485)
- RTC is standard (optional on KD485-PROG)
- all serial ports share the same ground (P1-P2 isolated on KD485-PROG)
- LEDs (except PWR) are user writable

- debug output via USB VCP (internal TTL-level serial on KD485-PROG)

Completely new features include

- Ethernet, with TCP/IP stack and TLS
- USB
- RTOS
- analog sensor interface
- 0-20mA current sink for 4-20mA sensor emulation/ relay drive
- FLASH file system (also accessible via USB) for data and config storage
- expansion options (ADCs DACs and anything on SPI)
- ARINC429 and other factory options

Hardware Description



The RS232 option on Port 3, and the hardware handshake mode on Port 2 are not shown above. The RS232 option on Port 3 is always fitted with the RS232 option on Port 4 and can co-exist with the CAN option.

Power Supply

This is an isolated switch-mode power supply, with a 10-25V input range. It generates individually isolated regulated outputs of +5V, +3.3V, and -20V for energising a 4-20mA analog sensor. It is a high efficiency design which uses advanced amorphous metal magnetics and its extreme reliability has been proven over tens of thousands KD485-family products since 1995. The -20V output for 4-20mA sensors needs a 24V input.

The KDE485 can also be powered via the USB connector (except the optional isolated analog subsystem). The Host should be capable of supplying 500mA. Typical current draw is 70mA.

Processor

This is an ST 32F417 ARM-based microcontroller running internally at 168MHz. 1MB of internal FLASH supports user applications of up to almost 1MB total size. A factory option of a 32F437 increases RAM by 64k.

4MB of external serial FLASH is also included for data storage, and a 2MB FAT file system.

Serial Ports

Four standard serial ports (2x RS232, 1x 2 wire RS485, 1x 4 wire RS485 or RS422 - optionally 4x RS232) are provided. See the **Serial Ports** section for details.

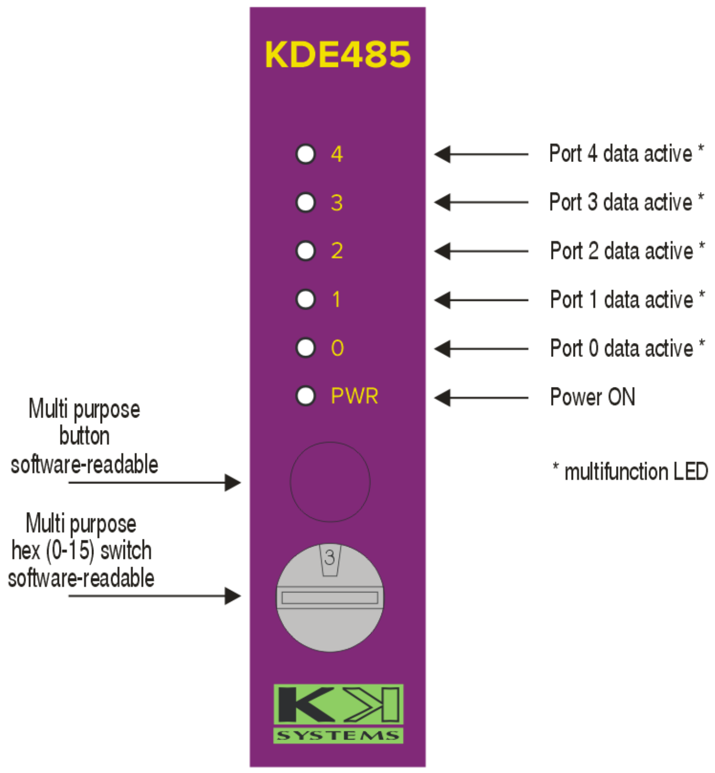
Ethernet

This is implemented internally on the 32F417, with 10/100 mbps, auto-detect, and auto-MDIX. A separate PHY controller is used and the RJ45 connector has integrated magnetics. As is standard with Ethernet, it is isolated.

USB

This is implemented internally on the 32F417. It is Full Speed USB (12mbps) client. KDE485 software supports the 2MB FAT filesystem (MSC device), and a virtual COM port (CDC device).

Indicators and Switches



The front panel contains a Power LED and five code-accessible LEDs 0-4. A "USB active" LED is next to the USB connector.

LED functions 0-4 depend on running application. The default mode (led_comms=1 in config.ini) is that LEDs 0-4 indicate data flow on ports 0-4 (serial ports 1-4, 0=USB VCP).

A pushbutton switch is user software readable. In addition:

- it is used with the hex switch to enter codes at boot-up; see **Factory Software Recovery**.
- a press > 2 seconds unmounts the USB drive (flushes USB Host cache)
- a press > 10 seconds with hex switch set to other than 0 or F reboots the KDE

Last two functions above operate only after KDE485 has successfully started-up, and they need the hex switch be set to a value other than 0 or F. The 0 setting is used to reset the KDE485 and capture 4-digit codes. The F setting is used to reset the KDE485 and bypass the watchdog, allowing a KDE485 to be recovered if the watchdog is constantly tripping.

The 16-position hex switch is user software readable.

Standard 0-20mA Current Sink

An analog current sink of 0-20mA is provided. The main application is enable the KDE485 to emulate a 4-20mA sensor. This also has a "relay drive" open-drain output whereby a relay coil (up to 24V, 100mA, external supply) can be energised. This function can be optionally factory built to output a 0 to +2.5V voltage instead.

Optional Isolated Analog Subsystem

This is an isolated analog interface, capable of directly reading a single sensor of the following type: PT100, PT1000, thermocouple (all 8 types), AD590, 4-20mA, 2 voltage inputs.

Optional ARINC429

This is a factory option. It is implemented with a highly integrated ARINC429 controller (HI3593) with two inputs and one output. Low speed (12.5kbps) and high speed (100kbps) are supported.

Source code examples are provided, for an easy creation of custom ARINC429 data converters.

When this option is installed, the external serial Port 4 connections are replaced with 1 x ARINC429 RX and 1 x ARINC429 TX. A second RX channel is available internally.

Internal ADC1, ADC2 and DAC1, DAC2

These are 12-bit devices integrated into the 32F417 CPU and are accessible on an internal connector. They are high speed devices, capable of sampling rates in the region of 1MHz. ADC1 is used for the power fail data save feature. DAC1 is used for the 0-20mA current sink and the relay drive feature.

SPI3 Expansion Port

Of the three 32F417 SPI controllers, **SPI3** is available for expansion with custom add-ons. To make the API functions for SPI3 thread-safe, mutexes are used, and the SPI3 controller is re-initialised each time a different SPI3 device is accessed. It can run at speeds up to 21mbps. The code for SPI3 can be found in KDE_SPI.c

RTC Real Time Clock

This is integrated in the 32F417 processor. It is supported at power-down with a 0.22F supercap which runs for around 3 days.

A standard Unix "tm" API is provided for setting and reading the RTC.

For Ethernet-connected units, or if internal GPS factory option is installed, the RTC is synced from internet time or from GPS time.

Watchdog

This is integrated in the 32F417 processor. The default configuration is retriggered from a 1kHz timer interrupt, with a 1600ms timeout , so a basic crash protection level is provided even if the user program does not use the watchdog.

KDE485 Reset (Reboot)

Software reset is performed with the KDE_reboot() function.

Hardware (front panel) reset is performed by holding down the pushbutton for > 10 seconds, with a code other than F set on the hex switch. Immediately after the reboot, the KDE485 enters the code entry mode (LED0 flashing) and pressing the pushbutton 4 times continues the reboot process.

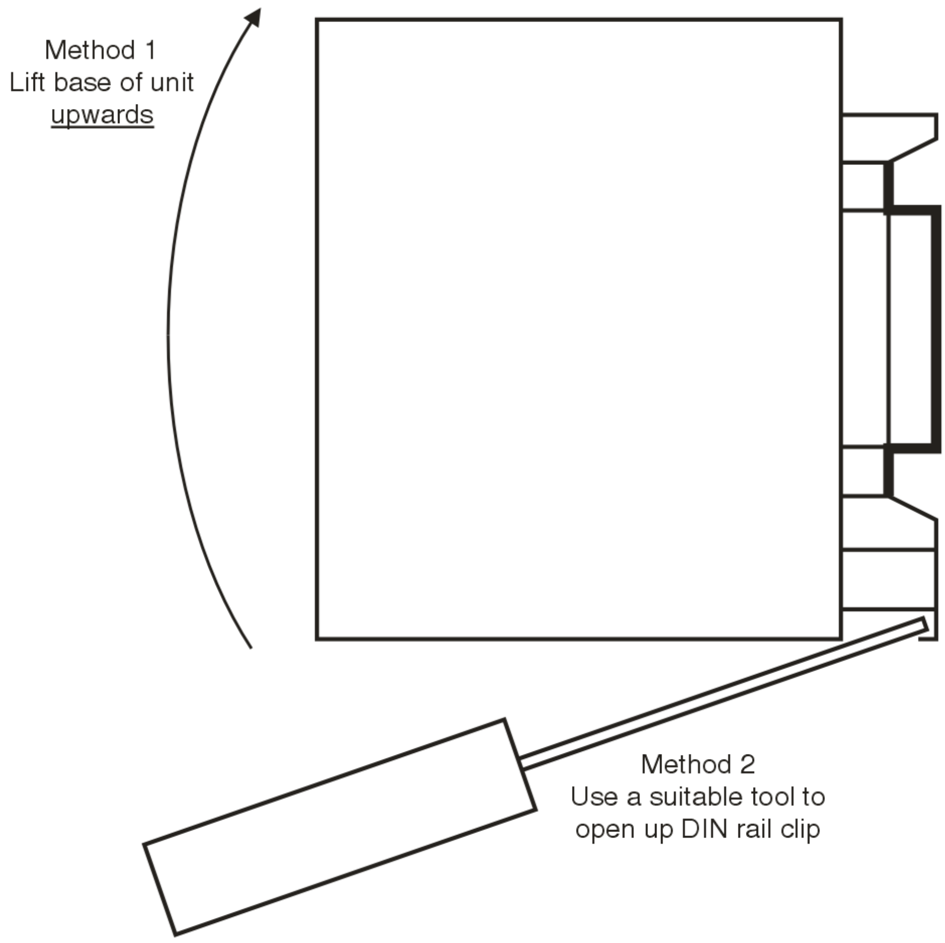
Optional CAN

This is a factory option. The 32F417 implements a CAN controller internally. No API exists currently. When this option is installed, some of the external serial Port 4 connections are replaced with CAN.

Optional SPI RAM (8MB)

This is a factory option. It enables block-oriented data storage and a read/write speed of around 2MB/sec. It is accessed in 512 byte blocks and is intended for data storage applications where e.g. FLASH would not be suitable due to wear.

Fitting KDE485 on and off the DIN rail



Software Description - Built-in Software

The KDE485 contains factory software which implements its functionality (the RTOS, filesystem, Ethernet, USB, software update management, TCP/IP, TLS, etc) and an API (application program interface) for user applications.

It is primarily intended for running user-created applications but it has some built-in applications so it can be used without writing anything:

Port-Port Serial Data Copy

This is a bidirectional data transfer application which forms 1 or 2 channels, each transferring data between any two ports. It is equivalent to KD485-ADE Mode 1. If a port is capable of half-duplex (port 3 or 4) then its driver is automatically enabled when transmitting. The baud rates etc can also be different. All five ports 0-4 are supported, with port 0 being the USB VCP (virtual COM port).

NTP Client

This sets the KDE485 RTC from an internet time source.

The alternative to an internet source is GPS, and a specially configured KDE485 is available as a GPS to NTP Server.

GPS to NTP Server

This application provides an NTP time server, over Ethernet, for networks where an internet connection is undesirable for security reasons. The internal SPI-GPS factory option, or an external RS232 or RS422 GPS, is required. A specially configured KDE485 is available as a GPS to NTP Server.

TCP to Serial Bridge

This application is a data-transparent (protocol independent) gateway between the KDE485's five serial ports (four physical ones and the USB VCP port), over a TCP/IP network. It creates pairs of interconnected KDE485 serial ports. The routing protocol is documented and this enables data connections between serial ports (e.g. Port 2) and TCP/IP ports (e.g. 192.168.5.77 port 10002).

The list of these applications is being continually expanded according to customer input.

Software Description - User Applications

The KDE485 is primarily intended to be a C-language user-programmable platform.

The C user application is normally developed using the STM Cube IDE development environment. This is available for multiple operating systems. See **Software Development on a PC**. Other environments can be used.

It is compiled to a file **firmware.dat** which is placed (via USB or via HTTP with an Ethernet-connected client browser) in the KDE485 2MB filespace. A reboot then processes this file, flashes it into the CPU FLASH, deletes it from the filespace, and restarts the KDE485 to execute the new code.

A debugger is thus not required, but an SWD debugger can be used for intensive development.

For accessing the KDE485 hardware features, a library of API functions is provided. These eliminate the need to know the hardware in depth and enable the creation of highly functional user applications with a very small amount of source code. See the **API** section for details.

This shows a simple KDE485 program which outputs "Hello World" every 1 second to serial port 3:

```
void vAPPTask(void *pvParameters)
{
	uint8_t string [] = { "Hello World" };
	while (true)
	{
		KDE_serial_transmit(3, string, sizeof(string));
		osDelay(1000);
	}
}
```

Software Development on a PC

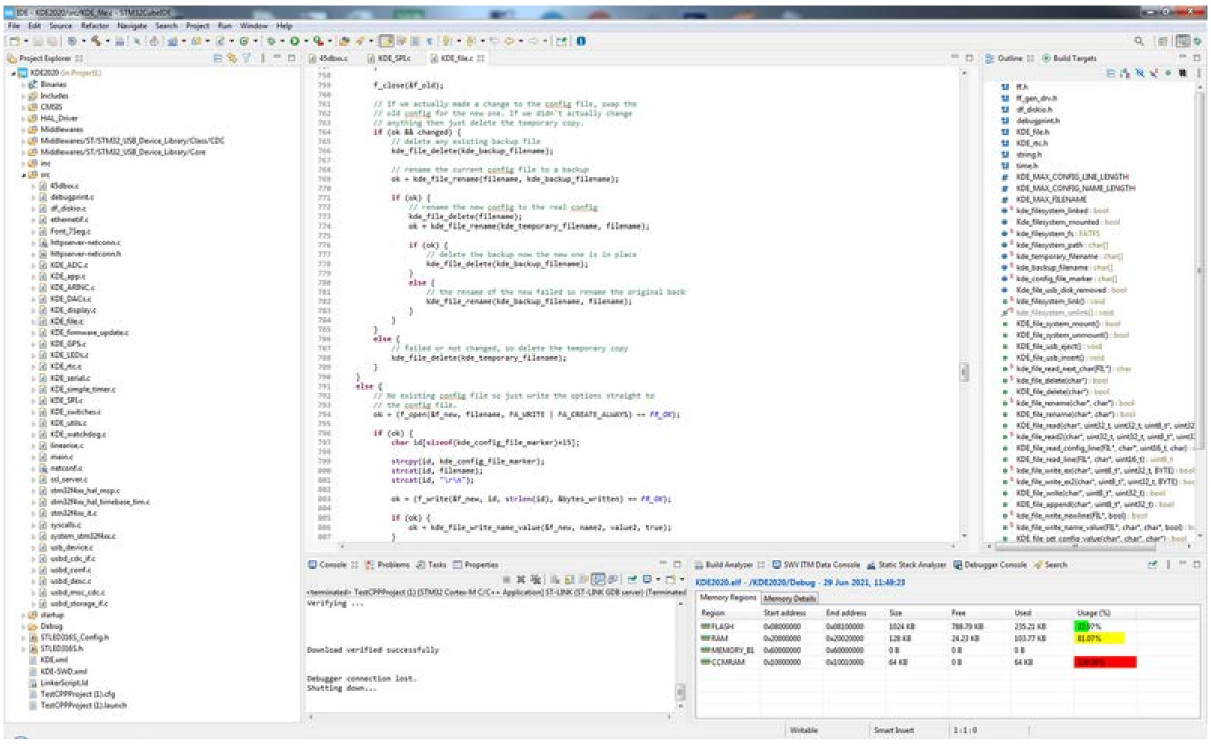
STM32 Cube IDE Overview

The KDE485 is programmed entirely in C and uses the STM32 Cube IDE programming environment. This is free from ST and is available for Windows, Mac OS and Linux:

	Part Number	General Description	Latest version
+	STM32CubeIDE-DEB	STM32CubeIDE Debian Linux Installer	1.12.0
+	STM32CubeIDE-Lnx	STM32CubeIDE Generic Linux Installer	1.12.0
+	STM32CubeIDE-Mac	STM32CubeIDE macOS Installer	1.12.0
+	STM32CubeIDE-RPM	STM32CubeIDE RPM Linux Installer	1.12.0
+	STM32CubeIDE-Win	STM32CubeIDE Windows Installer	1.12.0

No licensing is involved with Cube IDE and it can be installed on any number of PCs. Apart from a check for updates (which can be disabled) it does not go online which eliminates the risk of the tool suddenly becoming unusable. It also enables a complete project to be archived.

Cube IDE is based on the popular Eclipse IDE and supports all the functions expected in the modern software development environment: editing, compilation, and debugging.



```
        f_close(&f_old);

        // If we actually made a change to the config file, swap the
        // old config for the new one. If we didn't actually change
        // anything then just delete the temporary copy.
        if (ok && changed) {
            // delete any existing backup file
            kde_file_delete(kde_backup_filename);

            // rename the current config file to a backup
            ok = kde_file_rename(filename, kde_backup_filename);

            if (ok) {
                // rename the new config to the real config
                kde_file_delete(filename);
                ok = kde_file_rename(kde_temporary_filename, filename);

                if (ok) {
                    // delete the backup now the new one is in place
                    kde_file_delete(kde_backup_filename);
                }
                else {
                    // the rename of the new failed so rename the original back
                    kde_file_rename(kde_backup_filename, filename);
                }
            }
        }
        else {
            // failed or not changed, so delete the temporary copy
            kde_file_delete(kde_temporary_filename);
        }
    }
    else {
        // No existing config file so just write the options straight to
        // the config file.
        ok = (f_open(&f_new, filename, FA_WRITE | FA_CREATE_ALWAYS) == FR_OK);

        if (ok) {
            char id[sizeof(kde_config_file_marker)+15];

            strcpy(id, kde_config_file_marker);
            strcat(id, filename);
            strcat(id, "\r\n");

            ok = (f_write(&f_new, id, strlen(id), &bytes_written) == FR_OK);

            if (ok) {
                ok = kde_file_write_name_value(&f_new, name2, value2, true);
            }
```

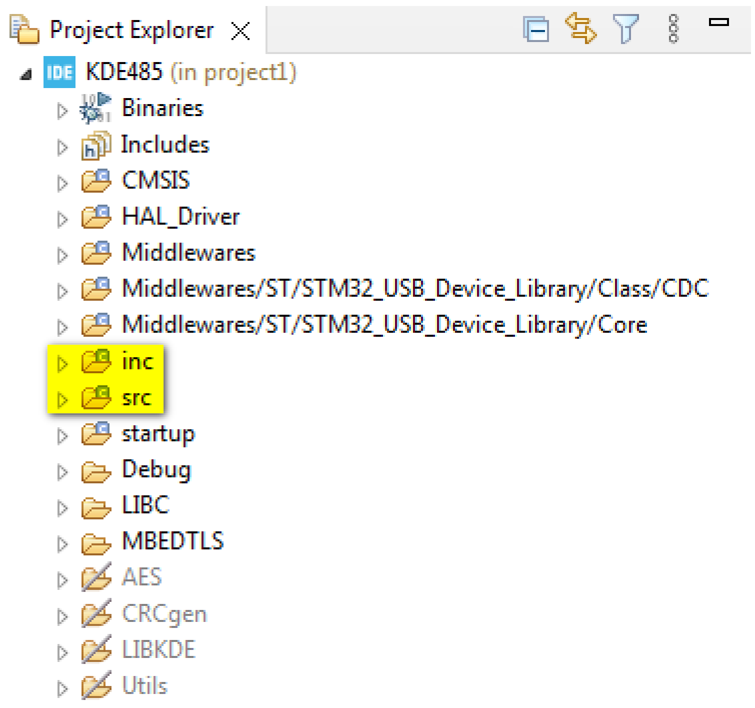
```
<terminated> TestCPPProject (1) [STM32 Cortex-M C/C++ Application] ST-LINK (ST-LINK GDB server) (Terminated
Verifying ...


Download verified successfully


Debugger connection lost.
Shutting down...
```

KDE2020.elf - /KDE2020/Debug - 29 Jun 2021, 11:48:23

Region	Start address	End address	Size	Free	Used	Usage (%)
FLASH	0x08000000	0x08100000	1024 KB	788.79 KB	235.21 KB	22.97%
RAM	0x20000000	0x20020000	128 KB	24.23 KB	103.77 KB	81.07%
MEMORY_B1	0x60000000	0x60000000	0 B	0 B	0 B	
CCMRAM	0x10000000	0x10010000	64 KB	0 B	64 KB	100.00%

There is also an STM project configurator and code generator called **Cube MX**. This is not applicable to the KDE485 and is not used. In this manual, "Cube" and Cube IDE" refers to the product formally called **STM32 Cube IDE**.

When Cube IDE is installed, and the sample project is imported, the entire KDE485 is visible as a Cube IDE Project - like this:



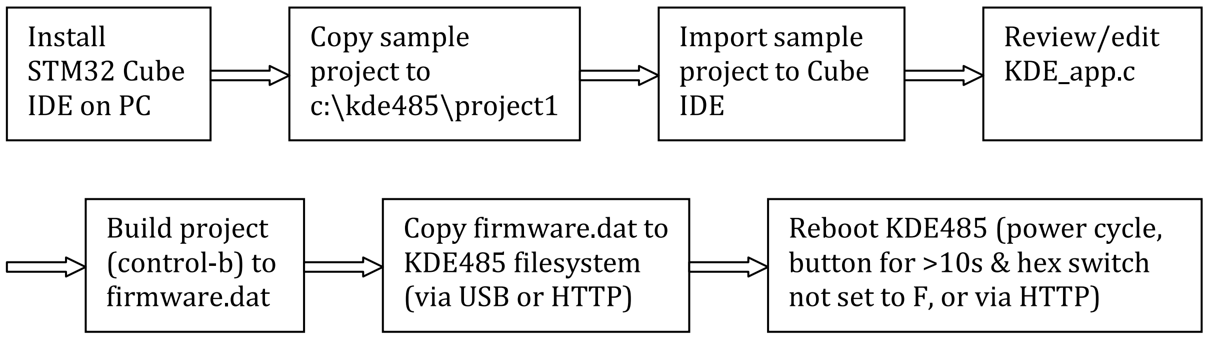
The sources to the software which specifically forms the KDE485 are mostly in the highlighted **src** and **inc** directories.

Most of the software is included in source form and the rest is provided as libraries and .h files.

To enable user programs to access the KDE485 hardware, a library of API functions is provided. These eliminate the need to know the hardware in depth.

The compiled file from Cube IDE is processed via a utility which appends a CRC to it. This CRC prevents programming the CPU with corrupted data. The file is then transferred to the KDE485 by placing it in the KDE485 filesystem (via USB, or HTTP) and power cycling the KDE485. The KDE485 detects the file when it starts up, checks the CRC, and flashes it into the CPU. The original factory firmware always remains stored separately in a reserved FLASH area and can be restored with a modified power-up sequence.

Getting Started with KDE485 Software Development

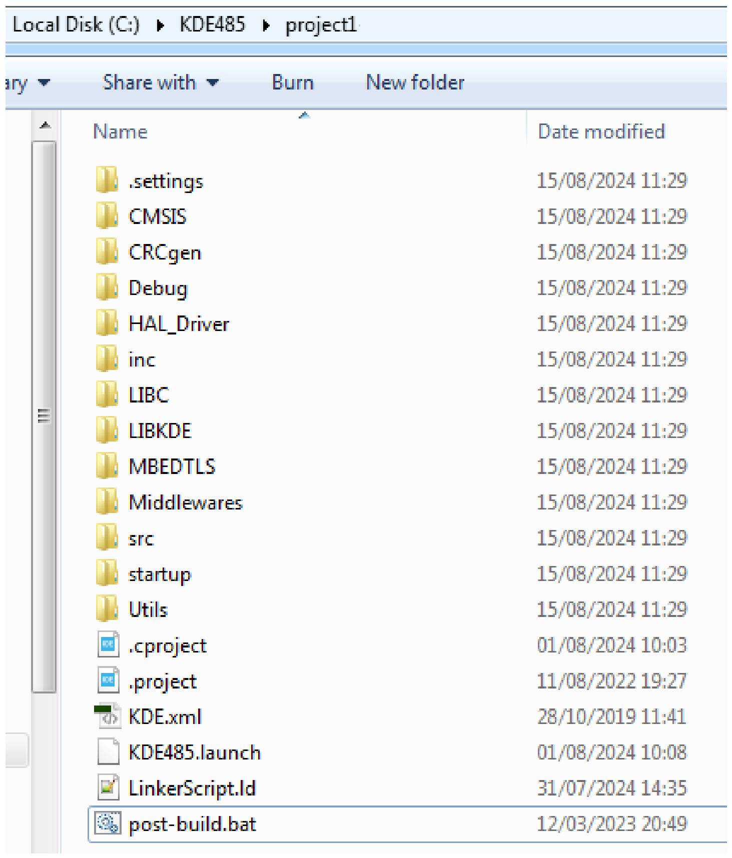

Install STM32 Cube IDE on PC
Copy sample project to c:\kde485\project1
Import sample project to Cube IDE
Review/edit KDE_app.c
Build project (control-b) to firmware.dat
Copy firmware.dat to KDE485 filesystem (via USB or HTTP)
Reboot KDE485 (power cycle, button for >10s & hex switch not set to F, or via HTTP)

Project Installation

This involves downloading the initial customer project development code.

The simplest way to set this all up is to create the directory (folder) structure of your project **before** starting Cube IDE, put the source files in it, and then install Cube IDE and point it to where the project is. We recommend using **c:\kde485\project1**; then instructions in this manual will make sense, and it will help to avoid obscure issues. The files can be found here https://www.kksystems.com/customer_area/project1.zip

Unpack the project1.zip files to the just-created c:\kde485\project1 directory on the computer. The directory should look like this


Local Disk (C:) ▸ KDE485 ▸ project1

ary ▾ Share with ▾ Burn New folder

Name	Date modified
.settings	15/08/2024 11:29
CMSIS	15/08/2024 11:29
CRCgen	15/08/2024 11:29
Debug	15/08/2024 11:29
HAL_Driver	15/08/2024 11:29
inc	15/08/2024 11:29
LIBC	15/08/2024 11:29
LIBKDE	15/08/2024 11:29
MBEDTLS	15/08/2024 11:29
Middlewares	15/08/2024 11:29
src	15/08/2024 11:29
startup	15/08/2024 11:29
Utils	15/08/2024 11:29
.cproject	01/08/2024 10:03
.project	11/08/2022 19:27
KDE.xml	28/10/2019 11:41
KDE485.launch	01/08/2024 10:08
LinkerScript.ld	31/07/2024 14:35
post-build.bat	12/03/2023 20:49

STM32 Cube IDE Installation - Windows

Cube IDE requires a minimum of Windows 7 64 bit, and requires Microsoft C++ redistributables 2019. It may run with redistributables 2015.

Prior to v1.8, Win 7-64 was listed as supported. V1.8 and V1.9 list only Win 8 and higher, and V1.14.1 and later list only Win 10, but all run fine under Win 7-64 and higher.

The latest version can be found at the STM website
https://www.st.com/en/development-tools/stm32cubeide.html

However, Cube IDE is an ongoing development by STM and new versions frequently introduce problems; usually fixed some months later. Therefore, **a specific version of Cube IDE (v1.14.1 at time of writing) is required** and the Windows installer can be found at
https://www.kksystems.com/customer_area/ST-Cube-IDE-1.14.1-windows.exe.zip

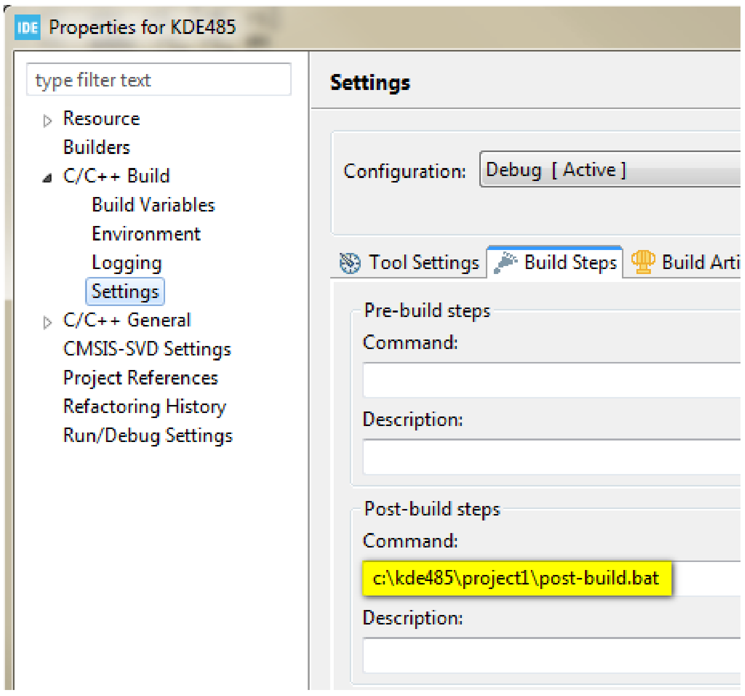
The installer has the ST signature on it, verified by Windows at installation time, so there is no security concern here.

If you have had a previous Cube IDE installation, uninstall it (in Control Panel) and then delete any c:\st directory.

On first startup, it asks for a workspace directory, similar to
C:\Users\user\STM32CubeIDE\workspace_1.14.1
The default setting is fine. It does not relate to where your project will reside.

Next, set up the post-build step post-build.bat. This performs tasks like adding a CRC to the generated binary file without which the KDE485 will not accept it.

Right-click on Project and under Properties enter the batch file (there are also options for relative paths):


Properties for KDE485

Settings

Configuration: Debug [Active]

Build Steps

Pre-build steps
Command:
Description:

Post-build steps
Command: c:\kde485\project1\post-build.bat
Description:

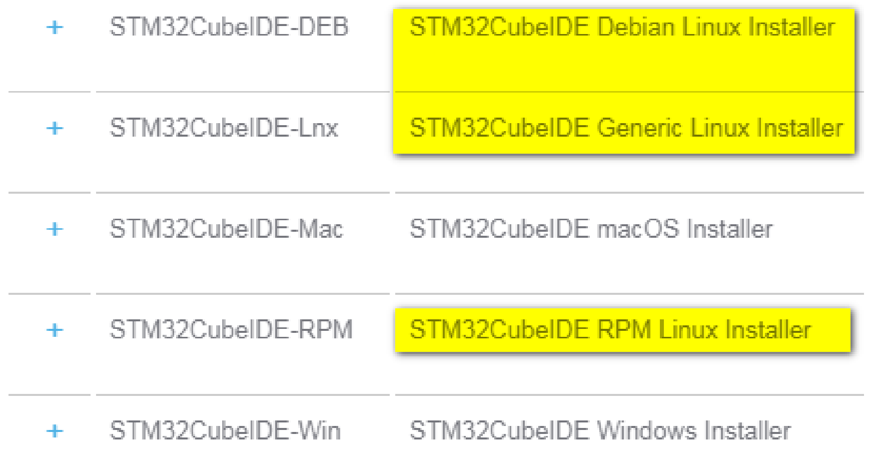
Click Apply and Close.

STM32 Cube IDE Installation - Linux and MacOS Installation

For information, please review the above instructions for Windows.

As for the Windows version, the KDE485 project is standardised at Cube IDE v14.

There are three Linux distributions and one MacOS distribution provided by STM:



+	STM32CubeIDE-DEB	STM32CubeIDE Debian Linux Installer
+	STM32CubeIDE-Lnx	STM32CubeIDE Generic Linux Installer
+	STM32CubeIDE-Mac	STM32CubeIDE macOS Installer
+	STM32CubeIDE-RPM	STM32CubeIDE RPM Linux Installer
+	STM32CubeIDE-Win	STM32CubeIDE Windows Installer

Version 14 of the above non-Windows installs can be found at

https://www.kksystems.com/customer_area/en.st-stm32cubeide_1.14.0_19471_20231121_1200_amd64.deb_bundle.sh.zip

https://www.kksystems.com/customer_area/en.st-stm32cubeide_1.14.0_19471_20231121_1200_amd64.generic-linux-sh.zip

https://www.kksystems.com/customer_area/en.st-stm32cubeide_1.14.0_19471_20231121_1200_x86_64.dmg-mac-os.zip

https://www.kksystems.com/customer_area/en.st-stm32cubeide_1.14.0_19471_20231121_1200_amd64.rpm_bundle.sh.zip

The Linux and MacOS Cube IDE installation differs from the Windows one, as follows:

The Windows CRC32 utility addcrc.exe is now a Linux "addcrc". Source code (addcrc.c) is included so you can recompile it for the other platforms. The Linux addcrc can be found below in both C source and a Linux executable. Also in the zipfile is the post-build .sh script which replaces the Windows post-build.bat file:

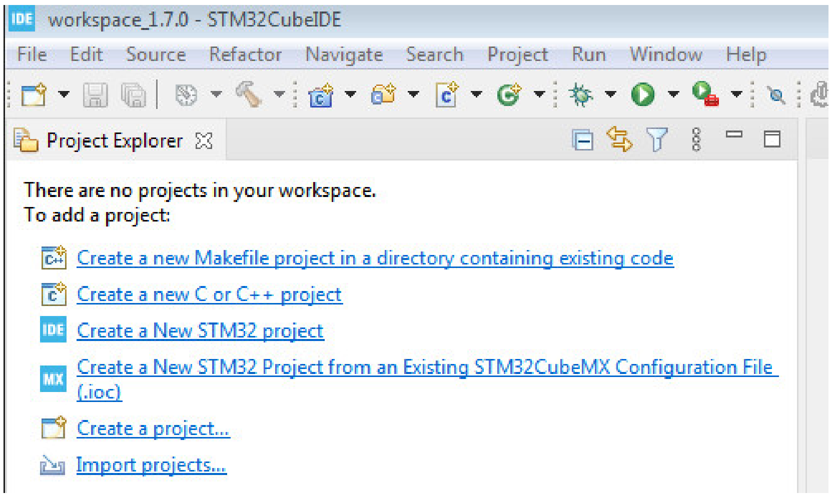
https://www.kksystems.com/customer_area/project1-linux.zip

Unzip the project-linux.zip file in the project directory, which will create post-build-linux.sh in that directory and addcrc in CRCgen directory. These two files should have the executable flag set, otherwise set it manually. Edit the project Properties (as for the Windows build) to run the post-build-linux.sh file as a post build step.

For ex-Windows projects, some build errors may appear, due to Linux being case sensitive.

Starting the new Cube IDE Installation

When a completely fresh installation of Cube IDE on this computer is started, it shows a general information/marketing form. Close this form. Then you see


workspace_1.7.0 - STM32CubeIDE

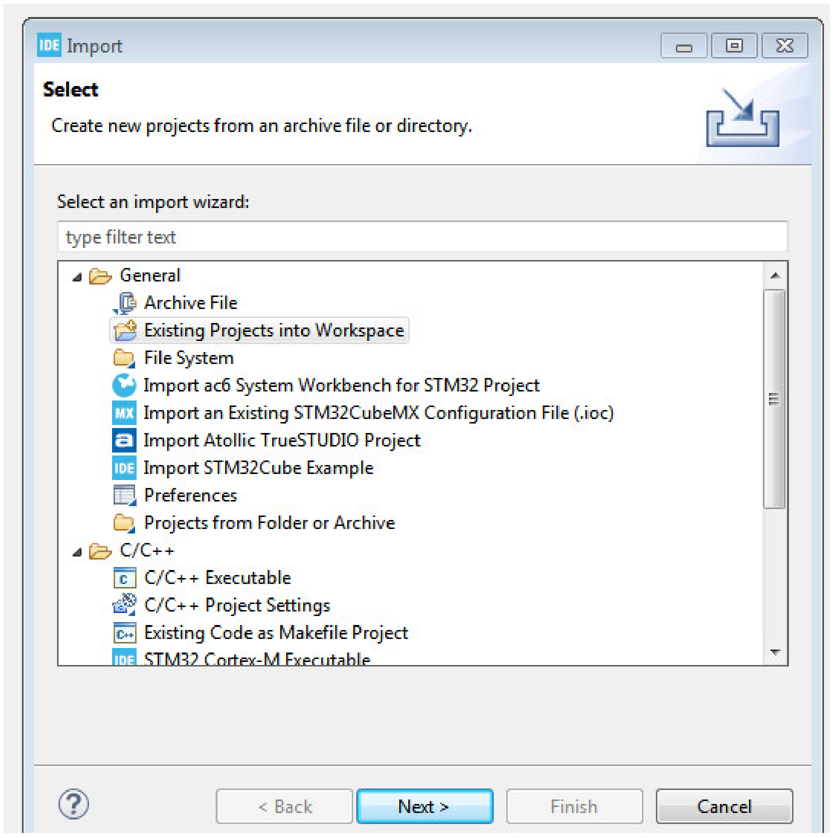
File Edit Source Refactor Navigate Search Project Run Window Help

Project Explorer

There are no projects in your workspace.
To add a project:

- Create a new Makefile project in a directory containing existing code
- Create a new C or C++ project
- Create a New STM32 project
- Create a New STM32 Project from an Existing STM32CubeMX Configuration File (.ioc)
- Create a project...
- Import projects...

Choose Import projects then Select / General and highlight Existing Projects into Workspace


Import

Select

Create new projects from an archive file or directory.

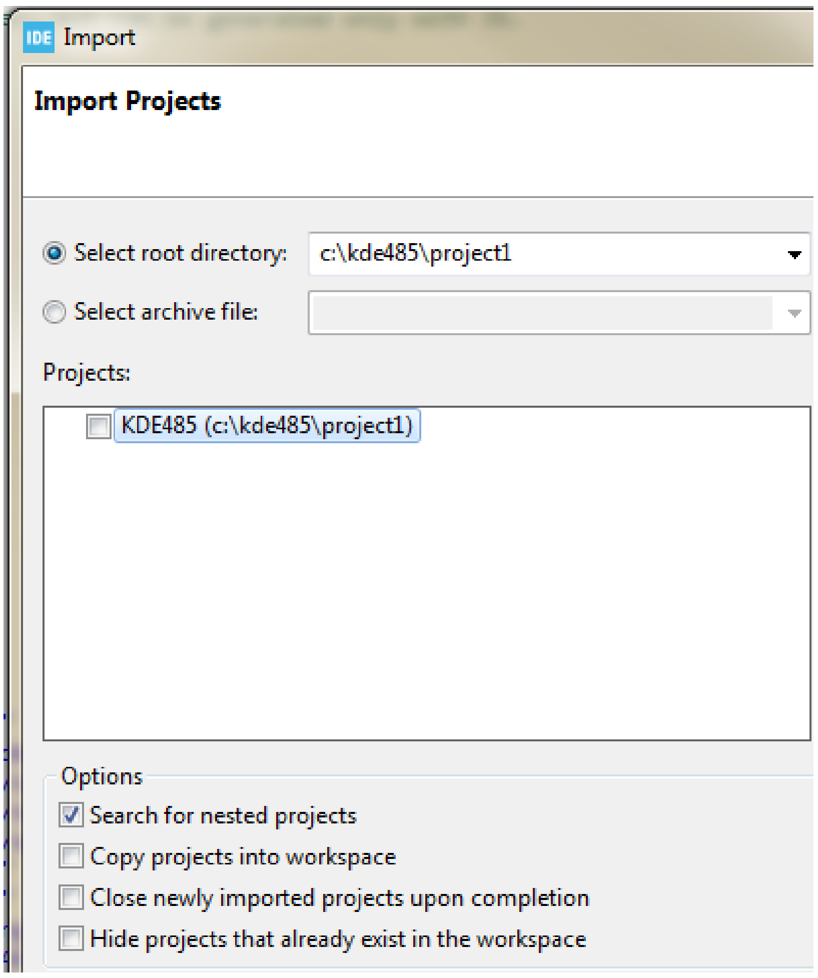
Select an import wizard:

type filter text

- General
  - Archive File
  - Existing Projects into Workspace
  - File System
  - Import ac6 System Workbench for STM32 Project
  - Import an Existing STM32CubeMX Configuration File (.ioc)
  - Import Atollic TrueSTUDIO Project
  - Import STM32Cube Example
  - Preferences
  - Projects from Folder or Archive
- C/C++
  - C/C++ Executable
  - C/C++ Project Settings
  - Existing Code as Makefile Project
  - STM32 Cortex-M Executable

< Back | Next > | Finish | Cancel

Next, specify your project location like this (check the checkbox next to the project name):


Import

Import Projects

● Select root directory: c:\kde485\project1

○ Select archive file:

Projects:

☐ KDE485 (c:\kde485\project1)

Options

☑ Search for nested projects

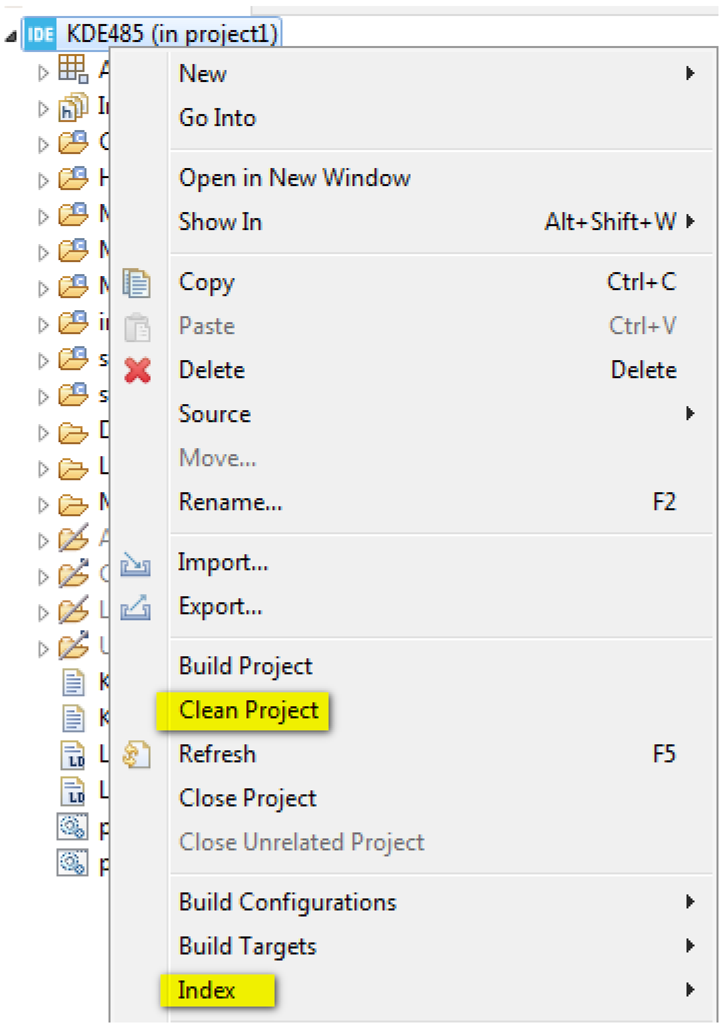
☐ Copy projects into workspace

☐ Close newly imported projects upon completion

☐ Hide projects that already exist in the workspace

Cube IDE then opens the project.

After importing the new project, the first step should be a right-click on the Project and Index / Rebuild. This is always a good idea when importing projects. It should be automatic but this may not work if files have been edited externally. Then run Clean Project.



The Clean Project usually fails the first time it is run; this is correct.

Cube IDE Project Build

Build Project (Build Project in the above menu, or control-b) will compile the whole project to a firmware.dat file in the Debug directory. Copy this file to the KDE485 filesystem and reboot the KDE485 (with a power cycle, a button hold for > 10 seconds and hex switch set to other than F, or via the HTTP server).

That is it! Your user program is now running. The demo application KDE_app.c contains a sample program which flashes the five addressable LEDs in a sequence. It runs as an RTOS task (see KDE_main.c for where it is started) so the rest of the KDE485 is still running.

At this point, it is a good idea to familiarise yourself with Cube IDE. It has a huge number of features, of which only a tiny proportion is needed. It is based on the Eclipse IDE (Integrated Development Environment) which is widely used.

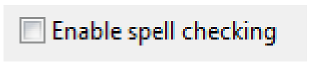
All Cube IDE features are available when developing for the KDE485, except the single-step debugging functionality which requires an STLINK or Segger debugger connected directly to an optional connector on the KDE485 circuit board.

Other Cube IDE Config Items

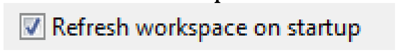
Cube IDE works mostly "out of the box" and most project preferences get set up by importing the sample project as described above. However the following general config items are worth doing at this point:

Window -> Preferences:

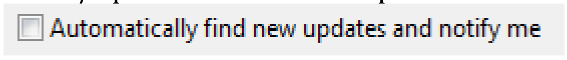
General -> Editors -> Text Editors -> Spelling -> disable


☐ Enable spell checking

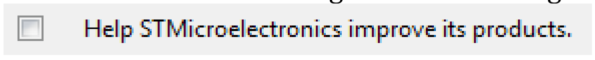
General -> Startup and Shutdown -> Refresh workspace at startup -> enable


☑ Refresh workspace on startup

Install/Update -> Automatic Updates -> disable


☐ Automatically find new updates and notify me

STM32Cube -> End User Agreements -> Usage stats -> disable


☐ Help STMicroelectronics improve its products.

The above configuration delivers a development kit which is standalone and not reliant on STM online services which may be discontinued or cause problems. In particular, updating Cube IDE also periodically updates the GCC tools and this will definitely break your project because of the tendency to change compiler or linker command line options default settings, etc. Later versions of GCC tools should work just fine but are not necessary and one spends a lot of time dealing with the issues mentioned above.

KDE485 Software Development - Advanced Information

This chapter contains extra information which is not immediately necessary to know.

When importing an existing project, importing any files edited externally, or after editing any files using an editor outside of Cube IDE, it is recommended that you always Index it and Clean it before building it.

Creating a duplicate Cube IDE project installation on another PC

The procedure is the same as for the original setup described above: Copy your project files into c:\kde485\project1 on the other computer, install Cube IDE, and import the project.

Cube IDE Post Build Steps

There is only one post build step for the KDE485 user program build: append a CRC32 to the binary file.

This is defined under Project / Properties / C/C++ Build / Settings / Build Steps / Post Build Steps and runs as a batch file:

```
"${ProjDirPath}/post-build.bat
```

This file contains the commands which generate the file firmware.dat, with the correct CRC32 appended. The CRC program is addcrc.exe. A Linux version is provided but if you are running a Mac OS version of Cube IDE then you will need to compile the provided addcrc.c source.

If you are loading code into the KDE485 with a debugger then you do not need the above post-build step (because the file firmware.dat is not needed; the debugger picks up the ELF intermediate file generated by Cube IDE) but it is useful because the file firmware.dat can be sent to a user in the field to update a KDE485 there, or it can be updated remotely via the HTTP server, or other forms of remote access.

Cube IDE Configuration, Compiler and Linker Options

There is a vast array of options but importing the supplied sample project will set it all up for you - except minor items like Cube IDE editor preferences.

Cube IDE uses GCC tools, which are currently (Cube IDE v 1.14.1) at version 11. Due to the popularity of the ARM32 architecture, these are high quality tools and there is nothing gained from upgrading them. In fact there is no point in any Cube IDE version later than 1.14.1.

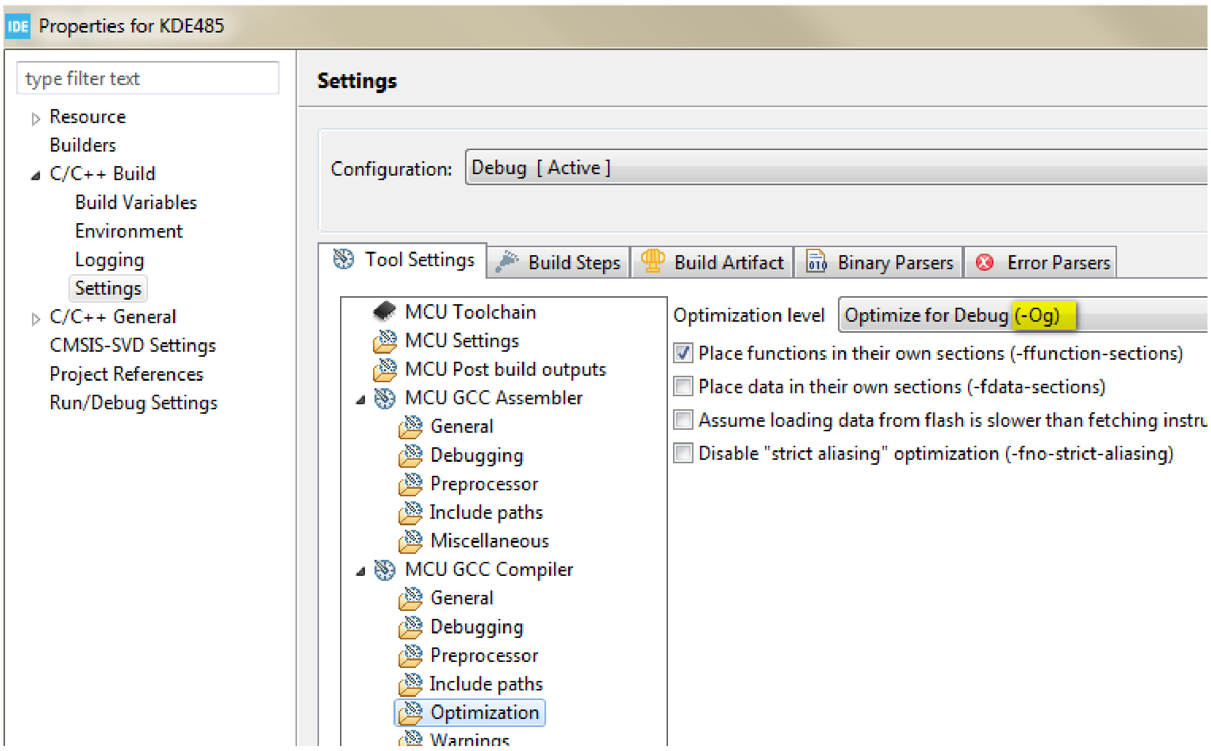
Compiler Optimisation

This is a vast topic over which coders get really excited! The GCC compiler supports various options, for smallest size, to highest speed, and some in between, and no optimisation at all. The following figures show the code size at a particular stage of one project

-O0 produces 491k
-Og produces 342k
-O1 produces 338k
-Os produces 305k

Others, not listed above, have been tested but make little difference.

Optimisation is set under Project / Properties


Properties for KDE485

type filter text
- Resource
- Builders
- C/C++ Build
  - Build Variables
  - Environment
  - Logging
  - Settings
- C/C++ General
- CMSIS-SVD Settings
- Project References
- Run/Debug Settings

Settings

Configuration: Debug [Active]

Tool Settings | Build Steps | Build Artifact | Binary Parsers | Error Parsers

- MCU Toolchain
- MCU Settings
- MCU Post build outputs
- MCU GCC Assembler
  - General
  - Debugging
  - Preprocessor
  - Include paths
  - Miscellaneous
- MCU GCC Compiler
  - General
  - Debugging
  - Preprocessor
  - Include paths
  - Optimization
  - Warnings

Optimization level: Optimize for Debug (-Og)

- [✓] Place functions in their own sections (-ffunction-sections)
- [ ] Place data in their own sections (-fdata-sections)
- [ ] Assume loading data from flash is slower than fetching instru
- [ ] Disable "strict aliasing" optimization (-fno-strict-aliasing)

Zero optimisation (-O0 (0=zero)) works fine, is easily fast enough for the job, but it produces about 30% more code than other levels. This increase may be partly due to a suppression of unreachable code removal. That is done by the linker and is extremely useful in any project where large libraries are a part of the project, have not been explicitly excluded from the build, but their functions are not called by anything.

The "general" level, -Og, is the recommended one for all KDE485 usage. The entire KDE485 has been built with -Og. This is shown in the screenshot above.

Any optimisation level should produce code which runs - as a **user** program which does not do direct hardware access - on the KDE485 but the higher levels can cause very hard to find problems; examples:

Optimisation levels above -O1 attempt to identify loop structures and can replace this loop

```
for (uint32_t i=0; i<length; i++)
{
    buf[offset+i]=data[i];
}
```

with a call to the memcpy() library function. This will crash the system if for some reason memcpy() is not accessible. In KDE485 user programming this scenario should never occur (because stdlib is always available) but there are other cases e.g. the memcpy() function is not guaranteed to support overlapping memory regions but your C loop may do so! There is also a possible performance loss since your C loop may be doing 32 bit transfers while the stdlib memcpy() may be a primitive one implemented with byte transfers. See memcpy_fast() in KDE_utils.c for an example of this. A compiler command line switch -fno-tree-loop-distribute-patterns has been added to help prevent memcpy etc substitutions globally but it is not 100% reliable.

Other problems can be much more subtle if e.g. driving hardware directly. KDE485 user code should not need to do this because the API covers all supplied hardware, but the option is available for hardware add-ons. For example the CS (chip select) timing on some devices can be violated if you developed with -O0 and then switched to -O3. The 32F417 cycle time of 7ns makes it quote possible to break something. Beware especially of code found on say Github which may have worked OK on some 16MHz CPU and even then only by luck!

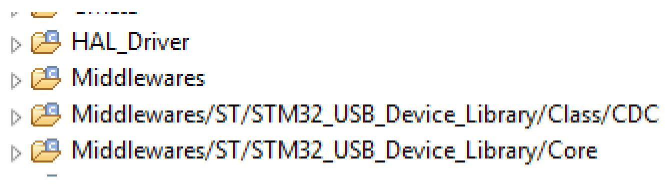
Changing the optimisation level under Project / Properties will not affect precompiled modules (basically everything for which the C source code is **not** provided) which should protect from hardware timing issues, but direct hardware access does exist in some modules for which the C source **is** provided (e.g. KDE_SPI.c) and that code can break. Substantial delays (of the order of 1µs) are done with KDE_hang_around_us() (in KDE_utils.c) and this is written in assembler and thus protected from optimisation effects, but this is not used for very short delays.

Finally, changing the optimisation level requires extensive regression testing which is almost impossible on any nontrivial product.

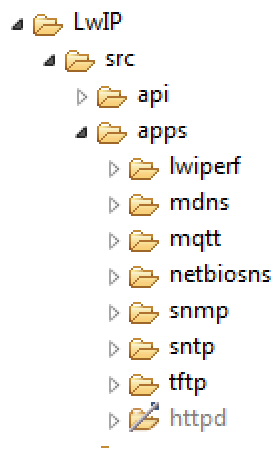
To obtain statistics on where your program is spending most CPU time, Cube IDE offers SWV Statistical Profiling.

Unreachable Code Removal

This is an important function of the GCC linker. Much of the ST supplied source which can be found in the supplied project tree, mostly here


HAL_Driver
Middlewares
Middlewares/ST/STM32_USB_Device_Library/Class/CDC
Middlewares/ST/STM32_USB_Device_Library/Core

and more specifically here

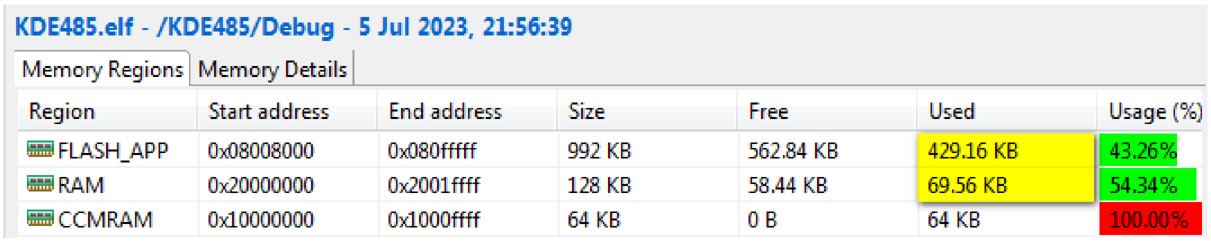

LwIP
src
api
apps
lwiperf
mdns
mqtt
netbiosns
snmp
sntp
tftp
httpd

is not currently called by anything in the KDE485 and will be stripped out. *The httpd module is excluded from the build under its Properties because the KDE485 has another HTTP server.*

The biggest software components on the KDE485 are ETH (everything to do with Ethernet) and TLS. These two can be disabled in KDE_netconf.h (or KDE_options.h in later KDE versions) with

```
#define INCLUDE_ETHERNET 1
#define INCLUDE_TLS  1
```

For example this is a normal Cube IDE Build Analyser display for something like the standard supplied sample project, showing a total of 429k of code usage and 69k of RAM usage:

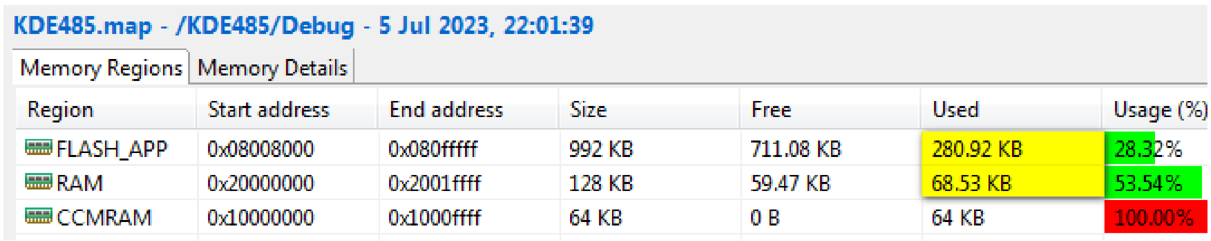


KDE485.elf - /KDE485/Debug - 5 Jul 2023, 21:56:39

Memory Regions | Memory Details

Region	Start address	End address	Size	Free	Used	Usage (%)
FLASH_APP	0x08008000	0x080fffff	992 KB	562.84 KB	429.16 KB	43.26%
RAM	0x20000000	0x2001ffff	128 KB	58.44 KB	69.56 KB	54.34%
CCMRAM	0x10000000	0x1000ffff	64 KB	0 B	64 KB	100.00%

With INCLUDE_TLS 0 the RAM usage is similar (because TLS temporarily allocates space on the heap) but the code usage has dropped dramatically from 429k to 280k:

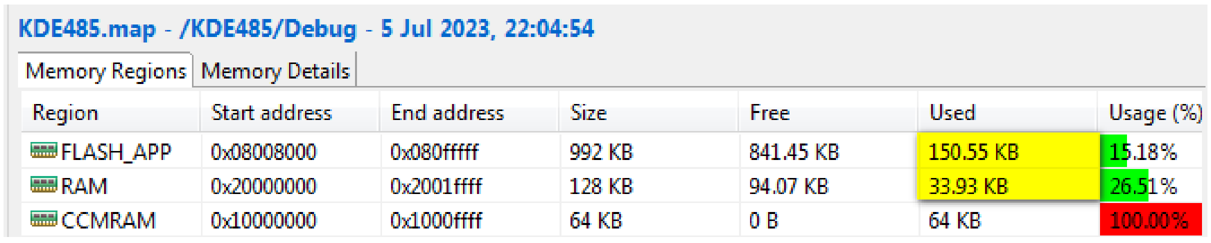


KDE485.map - /KDE485/Debug - 5 Jul 2023, 22:01:39

Memory Regions | Memory Details

Region	Start address	End address	Size	Free	Used	Usage (%)
FLASH_APP	0x08008000	0x080fffff	992 KB	711.08 KB	280.92 KB	28.32%
RAM	0x20000000	0x2001ffff	128 KB	59.47 KB	68.53 KB	53.54%
CCMRAM	0x10000000	0x1000ffff	64 KB	0 B	64 KB	100.00%

With INCLUDE_ETHERNET 0 the RAM usage has dropped from 68k to 33k (largely due to removal of ETH packet buffers and TCP/IP (LWIP) memory) and the code usage has dropped further from 280k to 150k:



KDE485.map - /KDE485/Debug - 5 Jul 2023, 22:04:54

Memory Regions | Memory Details

Region	Start address	End address	Size	Free	Used	Usage (%)
FLASH_APP	0x08008000	0x080fffff	992 KB	841.45 KB	150.55 KB	15.18%
RAM	0x20000000	0x2001ffff	128 KB	94.07 KB	33.93 KB	26.51%
CCMRAM	0x10000000	0x1000ffff	64 KB	0 B	64 KB	100.00%

The available heap space also improves in line with the gain in free RAM.

This code removal removes some occassionally surprising items e.g.

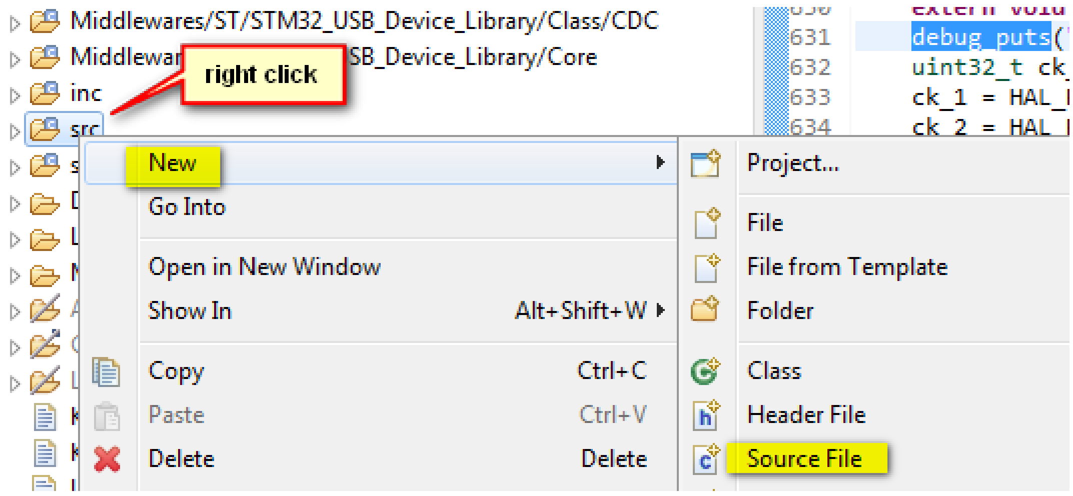
- ISRs without a vector
- unreferenced callback functions
- your entire program if nothing is calling main()
- code in a fixed address which is being called by calling that address

Debug and Release Builds

This is related to the above Optimisation topic. Cube IDE supports different build versions of a project. Traditionally these are called **Debug** and **Release** and can be configured under Project / Build Configurations. The general idea is that the Debug mode builds the project with zero optimisation and maximum debug symbol output, while Release builds it with maximum optimisation and zero debug symbol output. The problem with this is that you get your project working under Debug and then when you switch to Release various things may break due to issues.. Accordingly, the KDE485 Cube IDE configuration and the sample project **use only the Debug mode**. With the ARM32 tools, the binary does not contain debug information anyway; it is stored in other files, some of which are used by a debugger.

Adding New Source Files

When creating a new .c or .h file, you simply right-click on src or inc:



```
debug_puts(
uint32_t ck_
ck_1 = HAL_
ck_2 = HAL_
```

Cube IDE then automatically updates the makefile script according to the various dependencies.

Multiple Projects in Cube IDE

Cube IDE supports multiple projects. This functionality is not used for the KDE485 and is not recommended unless you are a Cube IDE/Eclipse expert. There is a risk of cross-referenced files and files ending up in other projects.

Version Control and Archiving

Cube IDE does not have version control. There are various version control tools which can be used, externally to Cube IDE. Preferences vary widely among programmers. The simplest way to backup your entire project is to backup the whole project1 directory contents and then you end up with a complete copy which is portable to any new installation of Cube IDE. The whole project can then be archived by archiving the project1 directory and the Cube IDE installer executable (around 200MB and 1GB, respectively).

Standard C Library

A complete standard C library is provided with the KDE485, covering the usual C functions like memcpy, the printf family, the scanf family, malloc and free, etc. The code is thread-safe and suitable for use under the RTOS.

Normally, a standard C library (whether the Newlib one popular in the ARM32 sphere, or the others) has various problems in the printf and heap family. It "works" on PCs where there is a huge amount of RAM. As one example, a "perfect float" algorithm, known as "Steele and White", became popular in the 1990s which uses a large amount of storage, and usually uses the heap, especially if %f or %l (float or long) formats are used, making it unsuitable for embedded systems. Another problem area is the heap itself (malloc and free) which is normally not thread-safe.

This has been solved in the KDE485, as follows:

1) An open source "printf" library, printf.c, https://github.com/MaJerle/lwprintf has been installed, and this replaces all of the problematic functions, with the following exceptions which remain in the original library, but aren't applicable to KDE485 because they are unix file stream functions:

svfprintf svfiprintf fiprintf vfiprintf

2) The above printf family replacement does not use the heap.

3) The heap (malloc and free) functions are mutex protected.

Furthermore, for special requirements, the standard C library functions can be replaced by user code on an individual basis. The library is libc.a and resides in the LIBC directory. To enable individual functions to be overriden with new code, it was processed with objcopy (see readme.txt and convert.bat in LIBC) to make all symbols "weak". The resulting file is called libc-weakened.a and this is a one-off conversion. The processed file resides in LIBC/LIBCW and is not affected by reinstalls of Cube IDE which would normally overwrite the libc.a file which resides in the c:\st\... tree. The linker configuration has been set up to pick up this library in preference to the original; this can be seen under MCU G++ Linker / Libraries where there are two entries, under Libraries and Library search path. Note the unix-traditional file naming convention where the "lib" leading part of the library file is stripped, and so is the .a suffix, so libc-weakened.a is specified as "c-weakened". The foregoing has already been done and should not need modification.

The final step is to place the replacement functions(s) into the src and inc directories (where e.g. printf.c and printf.h are), and adding your new .h file under MCU GCC Compiler / Include paths and there you can see

"${ProjDirPath}/inc/printf.h"

Use the Add button to include any new .h files there.

It is highly unlikely anyone will want to modify the above library but e.g. for specialised applications someone may want to replace the heap with a power of two heap.

Single and Double Floats

This is a performance gotcha for CPUs with hardware single floats but without hardware double floats. In the C language definition, floats are double unless defined otherwise. If you write e.g.

```
float x,y;
x = 0.5 * y;
```

the 0.5 is a double float so the multiply will use double float which on the 32F4 is a software implementation which is about 100x slower than the single float hardware multiply! You must use

```
x = 0.5f * y;
```

So, use the ‘f’ or 'F' after every numeric constant where the much faster single float maths is desired.

The above is irrelevant when the result goes to any of the printf() family because they defined as double by C convention. But if you use floats to get an easy dynamic range - a common scenario in engineering - and are not using the sscanf or printf type functions, your code will run much slower that it would otherwise. Single floats deliver 24 bits of resolution which is ample for the vast majority of real-world values.

Division by Zero

This is practically never useful and needs to be prevented by program design.

For **integers** this can be trapped into HardFault_Handler() and then into irq_handler_hard_fault_c(). These can be found in stm32f4xx_it.c. This is a unrecoverable trap. This trap is currently **not** enabled. It is controlled by SCB->CCR which is loaded with 0.

For **floats** this not trapped and the result is "inf" (infinity) which is defined within the IEEE floating point specification. Note that if your code produces a "zero" float, this could equally be +0.000001 or -0.000001, etc which will yield legal float numbers which will nevertheless be useless in the context of what you are trying to do.

Thread-safe Code Generation

Standard C does not guarantee code to be thread-safe (re-entrant) but the GCC compiler used in Cube IDE generates thread-safe code unless obviously unsuitable coding practices (e.g. static storage) have been used.

Interrupts

In most cases there will be no need to hook up interrupts, because creating an RTOS task is a much cleaner way to write code. However, it is supported:

The file vectab2.s (under Startup) contains interrupt vectors used in the customer code. For example

```
.word  TIM6_DAC_IRQHandler       /* TIM6 and DAC1&2 underrun errors */
```

is a vector taken for Timer 6, or the DACs. In the KDE485, timer 6 is used to generate the 1kHz timer tick (not Systick for the RTOS which is also 1kHz) which decrements various timers. The DAC interrupts are disabled. By convention, the handler for TIM6 is in stm32f4xx_it.c

```
void TIM6_DAC_IRQHandler(void)
{
      TIM6_ISR();               // HAL_TIM_IRQHandler(&htim6);
}
```

The actual ISR can be found in KDE_main.c

```
void TIM6_ISR(void)
{
      ADC1_power_fail_check();  // Check if 5V rail is dropping
      HAL_IncTick();            // increment a global variable "uwTick"
      Kde_simple_timer_tick();  // Process "simple timers" & LED port activity
                                // timers

      TIM6->SR &= 0xfffe;       // UIF=0 - clear interrupt pending flag
}
```

Sources are provided for most of the interrupt handlers. It is obvious how to hook up this 1kHz interrupt: insert your function before the last line above (the UIF=0).

The ARM32 architecture has relatively simple interrupt processing. An ISR is just a normal C function; there is no "interrupt" keyword, no RETI/IRET etc. All that is handled in hardware. However, enabling new interrupt sources, hooking them up, clearing them at source, etc, required a very careful reading of the CPU Reference Manual (RM0090).

Future versions of ST Cube IDE, Updates, Debuggers and Compatibility Limitations

Cube IDE comprises of several functions.

One is basically an editor and a makefile generator, and this portion of it should be usable with a KDE485 "for ever".

Another is the SWD debugger interface. This supports STLINK and Segger debuggers. STLINK V2 and V3 have been extensively tested. Segger J-Link Edu has been briefly tested and not found to be any better than the STLINK. The best is the STLINK V3 which supports the SWV ITM debug console feature which supports a specially patched printf() function in the KDE485 for debug output to a window in Cube IDE. The STLINK V3 ISOL offers isolation.

ST Cube IDE is updated every few months and - if you have updates enabled, which is **not** recommended - it offers to download the new version and install it over the top of the old one. No issues have been found with this process although it is "cleaner" to uninstall the existing one (via Control Panel / Programs and Features), download and install the new one from the ST website and re-import the project into it as described above under **ST Cube IDE Installation**.

However, at every few Cube IDE updates, Cube IDE installs new versions of the GCC tools - the compiler, linker, and other executables. A new version of the compiler will certainly generate different binary code. And it is obviously impossible for KK Systems Ltd to guarantee that some future version of these tools will not break something. Accordingly, when a new version of Cube IDE is released, always archive the current version of the Cube IDE installer .exe (about 1GB) before installing the new one and retain that version for as long as your product remains in production. This is a completely normal prudent practice anyway. The filename is of the form
en.st-stm32cubeide_1.14.1_20064_20240111_1413_x86_64.exe
The latest verified Cube IDE installation package can be found at
https://kksystems.com/kde485

The post build batch file provided writes out a file toolsversion.txt in the Debug directory which contains the version of the GCC compiler e.g.
arm-none-eabi-gcc (GNU Tools for STM32 11.3.rel1.20230912-1600) 11.3.1 20220712

Cube IDE Gotchas - General

Nothing is perfect, and Cube IDE does have some bugs, and strange behaviour.

The Build Analyser data sometimes doesn't show until you click on the .map file (found in the Debug directory). And sometimes the .map file is not created so a second ctrl-b is required.

If a source file has been changed externally to Cube IDE, or a new project has been imported, do an Index, Clean and Build to ensure Cube IDE is starting with a fresh set of files. With the configuration of the supplied Cube IDE sample project this should not be needed but is a good precaution.

Cube IDE Gotchas - With a Debugger

When you build a project in the Debug mode (e.g. F11, to load code via a debugger), Cube IDE captures the keyboard focus at the end of it, so if you were typing something in another application while building the project, a number of keystrokes may get captured into the current Cube IDE edit window, resulting in random "edits". You then need to close the affected file(s) with a "don't save" selection. **This could be dangerous.**

You can set a breakpoint where there isn't any code. You can set a breakpoint on a comment! This is just how the breakpoint functionality has been implemented, but often there simply isn't any code corresponding to the source file line. One case is removal of unreachable code (described earlier) and another is the ARM32 use of conditional instructions e.g.

```
if (x==9)
{
    y++;
}
```

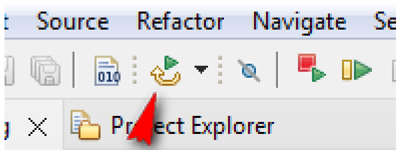
where a breakpoint on y++ may never get hit because it was implemented as a conditional *machine code* instruction, so the test-and-skip structure above doesn't actually exist. The solution is e.g.

```
if (x==9)
{
    asm("nop");           // set a breakpoint here
    y++;
}
```

Assembler code is **always** passed through to the binary code. This issue may not always be seen because compilers tend to implement the code conventionally if there would be more than around 4 conditional instructions in a row.

Cube IDE can eventually lose the connection to the debugger. It can take hours, or days. For long term testing it is better to output some debug text, instead of a breakpoint. This issue depends on the debugger type used. The STLINK V3 ISOL is the most reliable of the low cost units.

Cube IDE sometimes opens other files in the project when the execution is restarted e.g. with



The file which gets opened is where the CPU was executing code at the time of the restart. This is obviously undesirable but does no harm. This is another facet of the Debug mode issue above.

Running Cube IDE in a VM

It can be useful to run a KDE485 development system in a VMWARE VM, containing installed Cube IDE and the entire project for generating the firmware.dat file. Such a VM can be simply copied to any host machine which can run the free VMWARE Player. It is a great way to archive a project. The entire VM, containing e.g. Windows 7-64 and Cube IDE, with a generously sized virtual hard disk, is 20-30GB.

One can even run a USB debugger out of the VM if the USB port is appropriately configured. *It may not be possible to run a debugger out of the VM if the host machine is running Cube IDE also*. If not using a debugger in the VM, it is recommended to disable USB in the VM because it can affect the host machine's USB connection to the KDE485, whether a debugger is involved or not.

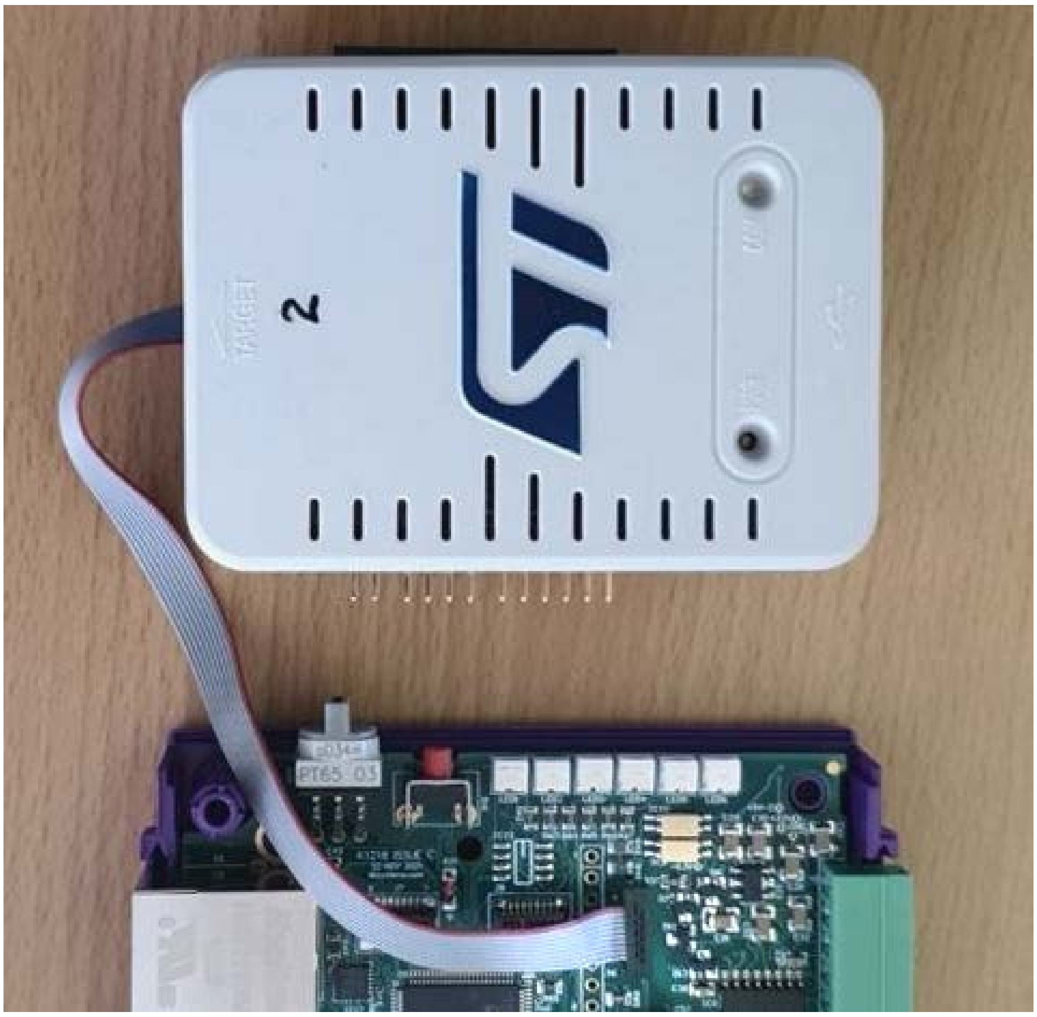
After a VM has been copied to a new machine, when it first starts up, it asks if it has been **copied** or **moved**. You must answer with **moved** otherwise it loses the Windows registration!

The minimum RAM in the VM for Cube IDE to run is around 3.5GB.

Using an SWD Debugger with Cube IDE

For intensive development projects, and where the debugging features of Cube IDE are required, the SWD debugging interface (similar to JTAG) is available on an optional internal 10-pin connector compatible with the ST-LINK V2 or V3 GDB (SWD) debuggers. Segger debuggers can also be used.

The KDE485 needs to be opened by removing the four visible screws. This shows the connection to an STLINK V3

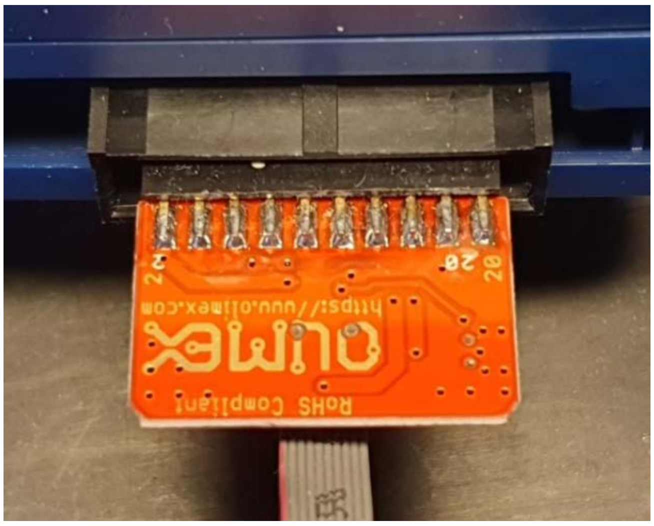


To load the user software into a KDE485, use the F11 build option. This loads the software into the CPU FLASH with the debugger, and runs it.

The STLINK V3 is available in an isolated (ISOL) version. This is the most reliable low-cost debugger. It also offers some protection from static and similar hazards.

The Cube IDE configuration for the debugger can be found under the SWV ITM Data Console even if that feature is not used.

The above photo shows a 10-way cable between the STLINK V3 and the KDE485. It has been experimentally determined that a more robust connection can be obtained (especially with STLINK V3 ISOL) by using the STLINK's 20-way connector with the Olimex " ARM-JTAG-20-10" adaptor to the 10-way cable:



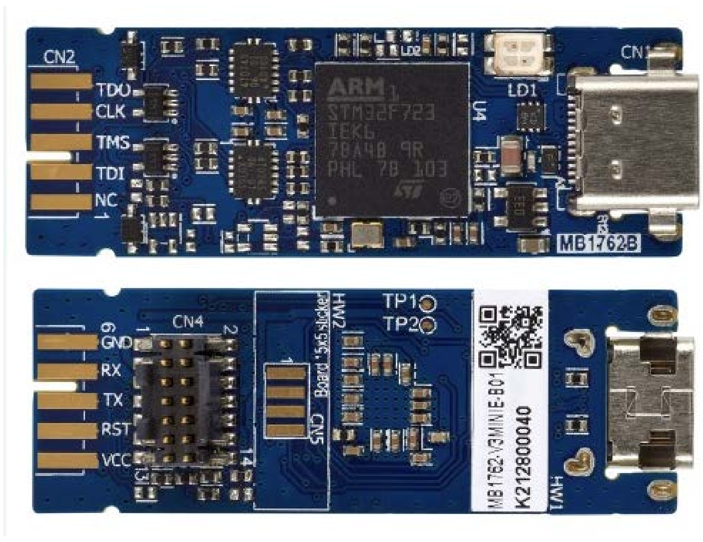
Olimex does not ship to some countries but these adaptors can be found on Ebay.

The older STLINK V2 ISOL can also be used

which does everything needed except the SWV ITM debugging console and other SWV debug functions. It needs the Olimex 20-10 adaptor mentioned above.

There are many other debuggers, including very cheap (under €10) Chinese counterfeits of STM products. These are best avoided because Cube IDE tends to recognise them and refuses to work.

There are also simple debuggers from STM e.g. the STLINK V3 MINIE



This debugger does work, with the supplued cable, but not for the Statistical Profiling function. It works for the SWV ITM debug console at 1000kHz SWO clock speed.

Higher up the cost scale there are Segger J-Link debuggers e.g. the Pro:

These have been tested to a limited extent and have not found to be better than the STLINK V3 despite costing 10x as much.

Recovering Factory Software

User software in the KDE485 is generated by creating RTOS tasks, generally starting with the sample KDE_app.c file.

User software running on the KDE485 has access to all the standard KDE485 functionality like Ethernet, HTTP server, USB (MSC and VCP), firmware upgrade capability, etc. To achieve this, the "development kit" contains the entire factory software, minus some portions e.g. the boot block. And most of the factory software is supplied in source form. Much of it is open-source e.g. FreeRTOS.

User software should never accidentally or intentionally disable USB operation and USB access to the filesystem. *By design, USB cannot be disabled in config.ini. Filesystem access can be retained via the HTTP server, unless this, or even ethernet itself, is disabled in config.ini or with the*
#define INCLUDE_ETHERNET 0
build option in KDE_netconf.h.

However, software can crash and USB access may be lost as a result of a programming error. Accordingly, the KDE485 implements a procedure for restoring the factory software:

Set 0 on the hex switch and hold down the button at power-up. Alternatively, use the 10-sec button press with 0 on the hex switch (this needs a running unit). After power-up, or when the 10 second period has elapsed, LED0 flashes rapidly while the button is held down. Now release the button. LED0 is again flashing rapidly.

The KDE485 is now in a mode where 4-digit code can be entered. The code entry is done by selecting each digit with the hex switch and briefly pressing the button. Each button press is confirmed with a flash. There is no timeout. When all four digits are entered, you get a double flash. If the four digit code is recognised, the selected action is taken. If it is unrecognised, the KDE485 continues the boot-up.

To abandon the code entry, enter 0000. This will never be a recognised code.

The factory software restore code is 1234.

The KDE485 now reboots and commences the CPU FLASH reprogramming with the factory software. After this it reboots and runs.

Setting Higher Levels of CPU Code Security

The 32F4xx CPU supports security modes which - for example RDP1 or RDP2 - disable SWD debugging and other external access to the CPU FLASH. These modes can be set via a debugger, or via software instructions.

Activating these features is irreversible and prevents the correction of bugs (in any part of the code - user code or factory code) and therefore it is entirely at user's risk and **voids any KK Systems Ltd warranty or liability (hardware or software)**. This is universal in the CPU business; setting any "security" which cannot be software-cleared voids the CPU manufacturer's warranty and any liability for the chip.

Also be aware that no security is perfect and all CPUs in common usage have known vulnerabilities in their "security" modes. The KDE485 thus cannot be regarded as a secure platform.

KDE485 Programming Information

The following sections describe the KDE485 capabilities in detail.

The various software and hardware features are described, each with its applicable API calls.

User programs are created without a need to access the KDE485 hardware directly.

Unless documented otherwise, all API functions are thread-safe i.e. they can be used by different RTOS tasks, and are mutex protected as appropriate.

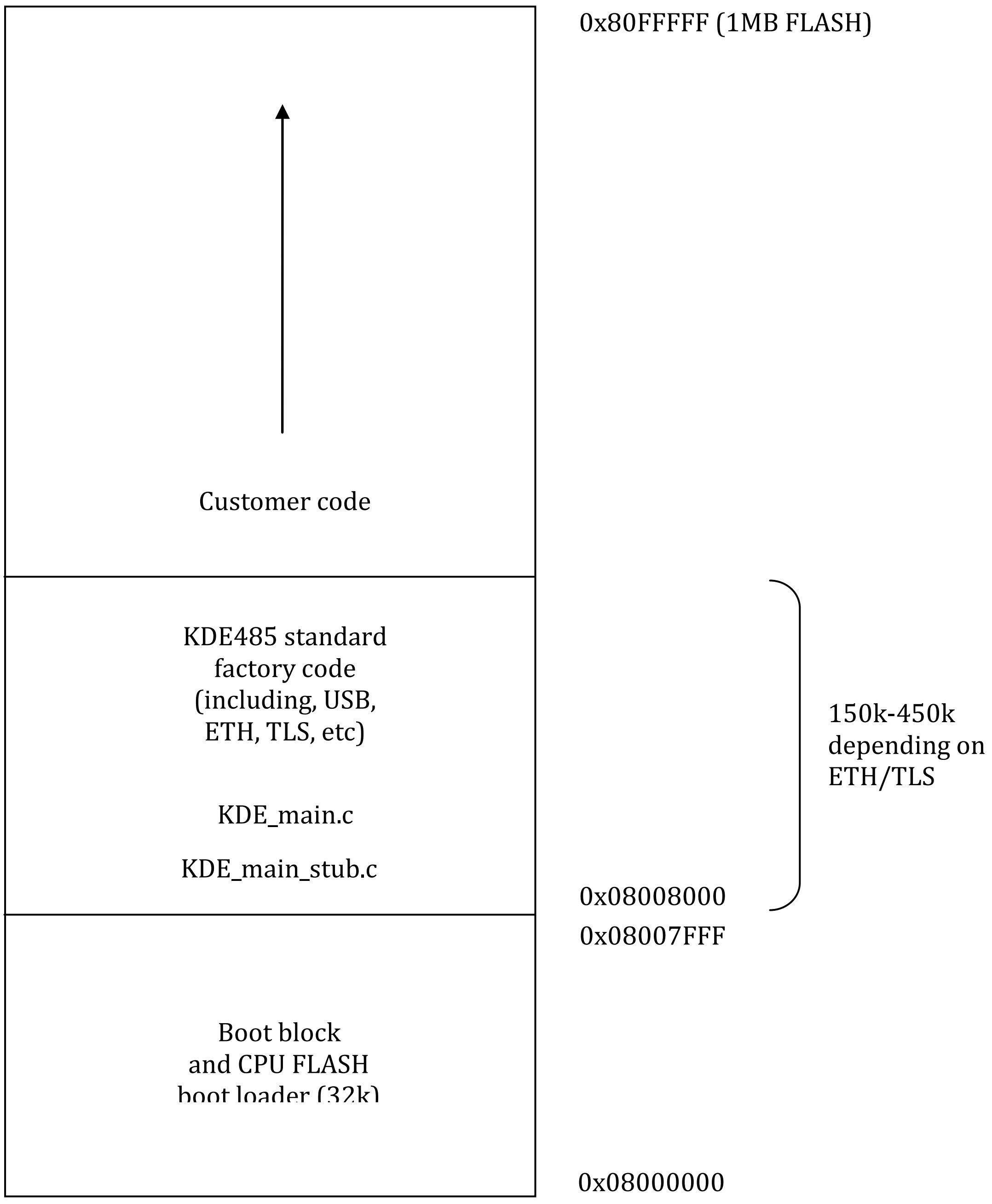
KDE485 Memory Map

The following is useful background information. It is not immediately required for programming the KDE485.

Memory Map - FLASH

The KDE485 memory map, both FLASH and RAM, is the "classic" one used in embedded systems. The following depicts the 1MB FLASH which applies to both 32F417 and 32F437.

Execution starts at the base of FLASH memory (0x08000000). The boot block, after various initialisation and other tasks transfers control to the customer code at KDE_main_stub.c at 0x08008000.

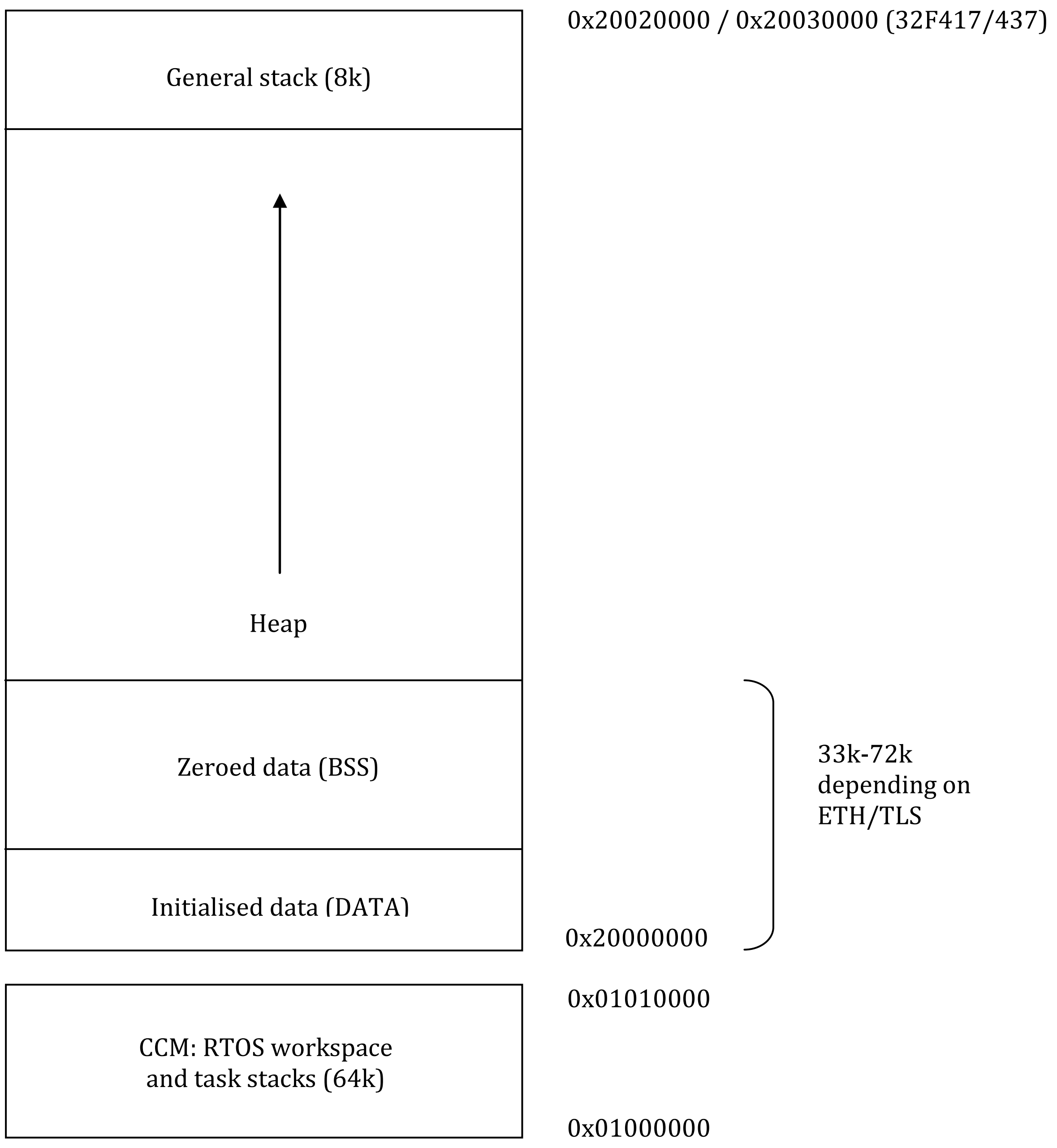


0x80FFFFF (1MB FLASH)
Customer code
KDE485 standard factory code (including, USB, ETH, TLS, etc)
KDE_main.c
KDE_main_stub.c
150k-450k depending on ETH/TLS
0x08008000
0x08007FFF
Boot block and CPU FLASH boot loader (32k)
0x08000000

Memory Map - RAM

This is the standard RAM memory map produced by common C compilers and linkers.

The following depicts the 128k or 192k RAM according to whether the CPU is a 32F417 (standard) or a 32F437 (optional).


0x20020000 / 0x20030000 (32F417/437)
General stack (8k)
Heap
Zeroed data (BSS)
33k-72k depending on ETH/TLS
Initialised data (DATA)
0x20000000
0x01010000
CCM: RTOS workspace and task stacks (64k)
0x01000000

The 64k RTOS workspace is in a separate memory area called CCM ("core coupled memory") which is faster than main memory but can't support code execution, DMA transfers, or the VTOR vector table. Locating the RTOS workspace in the CCM was a design decision; it could have gone into the main memory instead, and the CCM could have been used for e.g. the heap but that is wasteful if the heap is not used, etc. Experience has shown this decision to be reasonable because RTOS memory usage tends to be small (so the 64k is plenty) and any larger structures (buffers, etc) can be located in main memory by declaring them as static.

RTOS and Memory Allocation

The RTOS is FreeRTOS https://www.freertos.org. It is preconfigured in the KDE485 Cube IDE development kit and is very easy to use. API functions and sample code are provided for adding processes to the process list.

To create two RTOS processes:

```
xTaskCreate(vProcess1, (portCHAR *) "Process1", configMINIMAL_STACK_SIZE, NULL,
osPriorityLow, NULL);

xTaskCreate(vProcess2, (portCHAR *) "Process2", configMINIMAL_STACK_SIZE, NULL,
osPriorityLow, NULL);

void vProcess1(void *pvParameters)
{
   debug_thread_printf("Hello Process 1");
   osDelay(1000);
}

void vProcess2(void *pvParameters)
{
   debug_thread_printf("Hello Process 2");
   osDelay(1000);
}
```

The above will output the string every 1 second, while the osDelay() function yields back to the RTOS so we are not wasting a CPU time waiting in a loop.

Most user applications can be written as if they "owned" the CPU, and the RTOS works transparently. The osDelay(ms) function provides a delay in ms, while yielding control back to the RTOS so a process which needs to wait is not simply wasting CPU time.

The RTOS is configured to be pre-emptive on a time slice basis even at the **same** priority. *The FreeRTOS config parameter configUSE_TIME_SLICING is set to 1*. However, to use the RTOS simply and effectively, it is important to write the code so it yields back to the RTOS whenever it is waiting for something, with taskYIELD() or osDelay(x).

Yielding to the RTOS is also good for CPU power consumption i.e. chip temperature. When the RTOS determines, at the current tick, that there are no tasks to run, it executes a Sleep instruction (WFI) which puts the CPU into a low power state, until an interrupt occurs: the next RTOS tick, or one of the system interrupts.

RTOS task priority is a much debated topic. For a non-expert, the best way is to keep it simple and write code so it does not depend on *relative* task priority. In reality, most projects can be written with just osPriorityLow (the lowest recommended) used for everything, and yielding to RTOS when waiting for something. The CPU is time-sliced at 1ms intervals.

Note that taskYIELD() yields only to same or higher priority tasks, so it should not be used (if your aim is simply to “spread CPU power more evenly”) except inside a task running at the lowest recommended (osPriorityLow) priority. OsDelay(x) blocks the calling task for x ms and enables **all** other tasks to run in the meantime. Note that if x=1 then the actual delay can be anywhere from nearly zero to 1ms so if you need a *minimum* 1ms delay, use osDelay(2).

A difficulty can arise where you have task(s) running which

- are CPU intensive
- are not yielding (no osDelay() used)
- you are unable to modify them (come as a library, or just a huge amount of source)
- they need a high priority for other reasons

In the KDE485 this can happen if TLS (HTTPS) is used. TLS takes around 3 seconds to compute the crypto for the session setup, and does not yield to RTOS. Interrupts always continue to be serviced. Other tasks will also run because the RTOS is pre-emptive, but they will run slower during this time. TLS runs at the priority level of the task calling it, which can be quite low, but the effect of it not yielding can be noticeable. See **TCP/IP Application Tips**.

How much stack does an RTOS Task need?

The starting point for RTOS task memory allocation is the above configMINIMAL_STACK_SIZE which is 512 words (2k bytes). The smallest amount of stack which works is configMINIMAL_STACK_SIZE/2.

Some system functions use DMA (e.g. those for reading or writing the serial FLASH (the "linear FLASH" area, or the FAT file system), Ethernet, USB) and these either use main-RAM buffers or contain code to handle both CCM and non-CCM buffers.

In general, C code uses very little stack, for standard things like function calls. Most significant stack usage in the KDE485 architecture is in buffers. If your RTOS task defines a 1k buffer, then straight away you need to allocate configMINIMAL_STACK_SIZE (2k bytes).

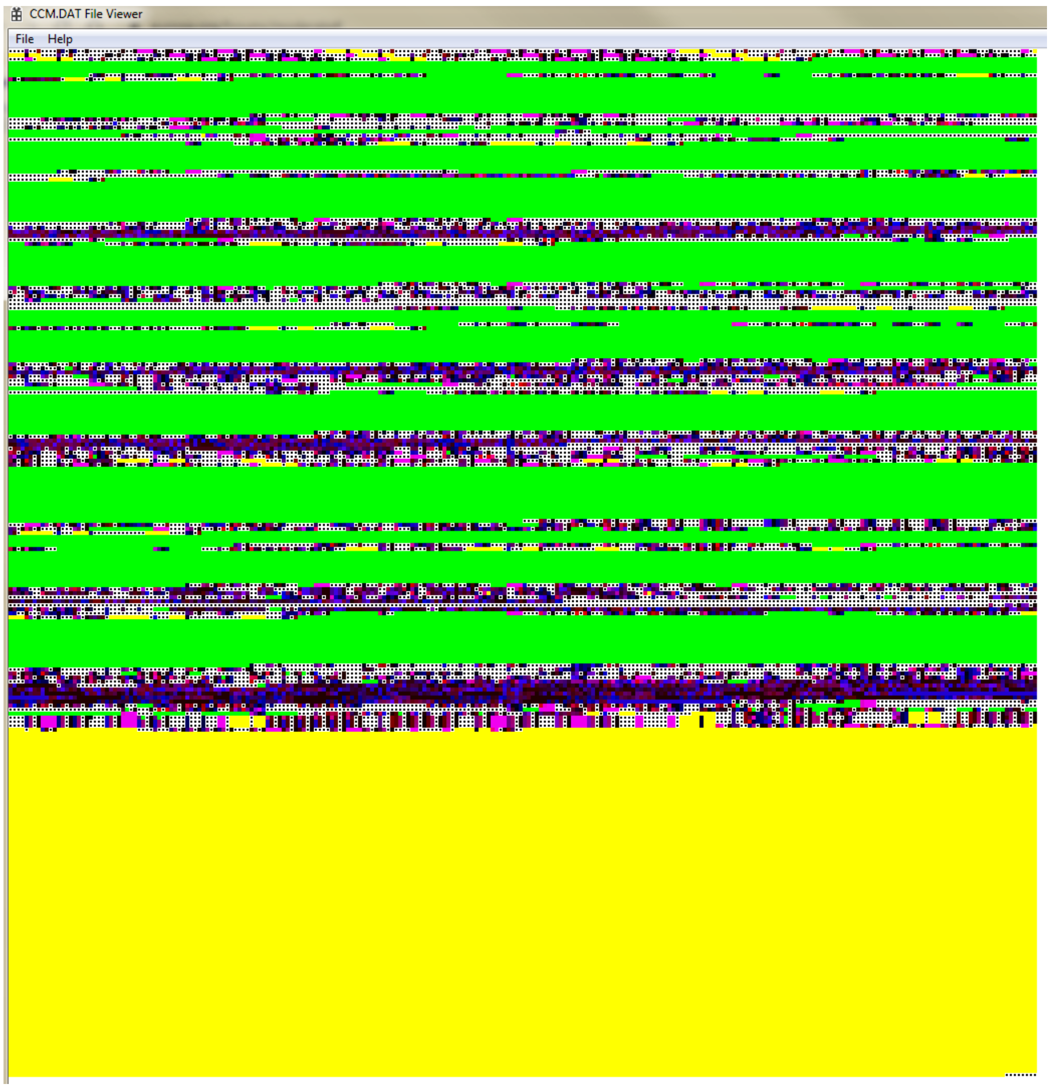
The 32F417 has two stack pointers. It switches to a second one to service interrupts, so ISR stack usage does not add to your code's stack usage.

As always in C, you can choose between storage on the stack (the default for any variable declared inside a function), and storage in main RAM (declared outside any function, or with the "static" keyword).

For RTOS tasks, stack storage is allocated in a dedicated RTOS stack space which uses a dedicated RAM section called the CCM ("core coupled memory") which is a higher speed RAM. CCM has some limitations (no code execution, no DMA access) but this is fine for the RTOS stack space. The CCM is 64k and approximately half is used by the system (for Ethernet and USB, mainly). If these processes are not enabled (in config.ini) most of the 64k is available for other RTOS task stacks. The RTOS stack usage is displayed in the HTTP Server Status page.

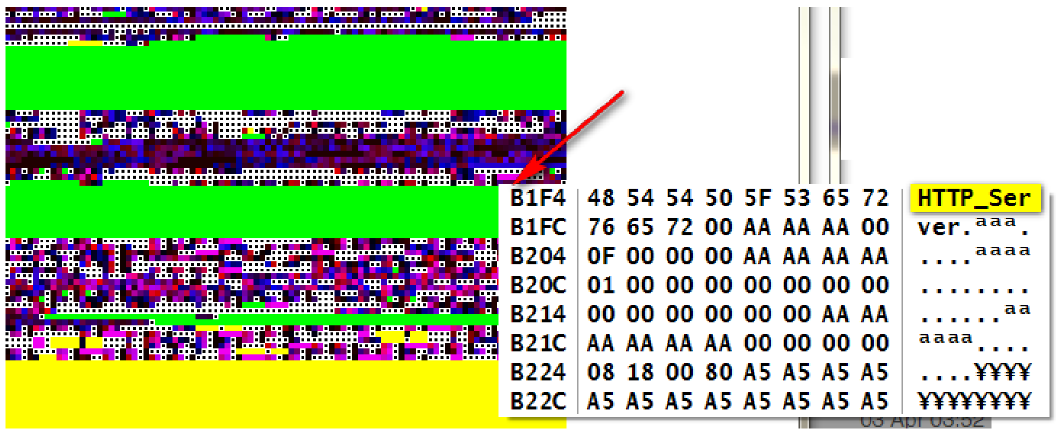
For a quick and easy overview of RTOS stack usage within the 64k CCM area, a Windows program CcmDatViewer.exe is included (in the Utils directory) which displays the 64k CCM area graphically, if the 64k CCM memory block is written to a file called ccm.dat.

```
The function
bool KDE_write_CCM (void);
creates this file in the KDE485 filesystem.
```



Green represents unused stack space within each RTOS task. As can be seen above, most allocation is generous. The task stacks are allocated as the tasks are defined. Yellow represents unused space before the end of the 64k block. At the beginning (top) are various RTOS workspace areas and message queues.

Within each task's stack you can find the task's name by hovering (while holding down the mouse button) near the end of it, as shown below



```
B1F4  48 54 54 50 5F 53 65 72  HTTP_Ser
B1FC  76 65 72 00 AA AA AA 00  ver.ªªª.
B204  0F 00 00 00 AA AA AA AA  ....ªªªª
B20C  01 00 00 00 00 00 00 00  ........
B214  00 00 00 00 00 00 AA AA  ......ªª
B21C  AA AA AA AA 00 00 00 00  ªªªª....
B224  08 18 00 80 A5 A5 A5 A5  ....¥¥¥¥
B22C  A5 A5 A5 A5 A5 A5 A5 A5  ¥¥¥¥¥¥¥¥
```

The Heap

The traditional C stdlib heap is available, via the standard malloc() and free() functions. The code originates from Newlib. These functions are mutex-protected and thread-safe.

The heap implementation uses 4-8 bytes (depending on block alignment) of metadata for each allocated block. It is thus not suitable for allocating thousands of small blocks. Execution time is around 10 µs for malloc+free.

The heap is **not** recommended due to system reliability issues resulting from fragmentation. Statically allocated storage is much safer. However, a heap is useful if fragmentation is guaranteed to not happen. This can be achieved by e.g.

- allocating a block at startup and never freeing it (some KDE485 optional features do this)
- freeing each allocated block before allocating another block
- a choice of block sizes versus available memory

On the last one above, e.g. allocating one 1k block and one 48k block within a 55k available heap space, and freeing these in any order, makes loss of memory due to fragmentation impossible.

The minimum allocatable size is 16 bytes and even malloc(0) returns a pointer to a block of this size.

There is no compaction or "garbage collection". Thus the address of a newly allocated block can vary, even if there is space for it at the base of the heap.

Note that the KDE485 actually runs three heaps:

- the "general heap" described above
- FreeRTOS runs its on private heap (within the 64k CCM area - allocated and never freed)
- LwIP runs its own private heap (within the MEM_SIZE memory area - currently 6k)
- MbedTLS runs its own private heap (within the TLS_MEMORY_SIZE area - currently 48k)

Only the first one is usable as a "general heap" although the FreeRTOS heap is used for RTOS task stack-based variables. The MbedTLS heap is thrown away each time a TLS session is closed.

The available general heap size at startup can be found in a global variable

```
uint32_t g_max_heap;
```

This returns the correct value regardless of which of the two CPU options is installed. If TLS is invoked, it temporarily allocates 48k of this during a TLS session.

Re-entrancy Considerations - is this "thread-safe"?

With the GCC compiler, most C code, with variables on the stack, is inherently re-entrant and is thus thread-safe. Static variables obviously are not, unless implemented as a 2-dimensional array where the 1st index is the number of the task (0,1,2 etc).

Hardware: One area where special care is needed is where hardware is being accessed. The same piece of hardware usually cannot be shared. In some cases it could be but with a cost; for example a multi-channel A-D converter could be read by multiple processes, but might as a result need to be completely re-initialised each time it is read. And obviously each conversion needs to be completed. It would need mutex protection around the complete initialisation+read cycle. *The SPI3 controller, which is used for multiple peripherals running at different clock speeds, has been implemented in exactly this way. If adding SPI3 devices, see KDE_SPI.c for examples.*

Inter-task comms: The RTOS offers various sophisticated devices for inter-task communication, but a simple way is to set flags, possibly with other data, in static memory. Care must be used that such flags are atomic. In GCC/ARM32 C, variables which are atomically read/written are: boolean, 8, 16 and 32 bit integers. Floats and 64 bit values are not; e.g. a uint64_t would be written in two 32-bit writes, and another RTOS task reading it could be task-switched in between the two writes. *LDRD and STRD do 64 bits atomically but a compiler won't generate these; you have to use assembler.*

Serial ports: The API functions for the four serial ports can be used by different RTOS tasks. A particular serial port cannot be meaningfully shared.

ADCs: The ADCs can be used by multiple tasks concurrently, with each one read by its own task.

DACs: The 12-bit DACs can be used by multiple processes concurrently, with each one driven by its own task.

Ethernet: This is accessed via LwIP which is thread-safe (for Netconn and Sockets API).

USB: This is more complicated. USB supports two profiles. MSC, for Windows access to the filesystem, which is not applicable here. CDC is used for the virtual com port. When used for debugs using the debug* functions, each debug message is "atomic" in that occupies a single line for clarity. When used as port 0, only one RTOS task can use Port 0.

ARINC429: The API functions for the HI3593 chip are mostly not thread-safe.

Front Panel Interface

LEDs

```
void KDE_LED ( int lednum, bool status );
```

This function turns the selected LED (0-4) on or off. This is for the user-addressable mode.

However, the five LEDs have the following system functions:

- indicate data activity on ports 0-4
- indicate GPS status

The system function is determined by the following config.ini values:

```
led_comms=0/1     ; 1 = system controlled
led_gps=0/1       ; 1 = system controlled
```

To enable the LEDs to be user-addressable, the above values need to be set to zero:

```
led_comms=0
led_gps=0
```

Alternatively this can be set at the start of the respective RTOS task with

```
g_led_comms=false;
g_led_gps=false;
```

See KDE_app.c for an example of the above.

Setting both values to a 1 produces an undefined result.

Hex Switch

```
uint8_t KDE_get_hex_switch (void)
```

Returns the hex switch 0-15.

Pushbutton Switch

```
uint8_t KDE_get_push_switch (void)
```

Returns the switch state: 1 = pressed.

FLASH Filesystem

Overview

The KDE485 has **4MB** of FLASH storage. This is allocated as follows:

2048k	FAT file system (FAT12)
1536k	Factory recovery code and other internal storage
512K	Linear data storage area

The 2MB FAT filesystem is accessible to both a user application program running in the KDE485 and via USB as a removable mass storage media visible to the USB Host (Windows etc).

Read speed varies from 300kbytes/sec to 2Mbytes/sec, depending on various factors.
Write speed is around 30kbytes/sec - the FLASH programming speed.

File System Limitations

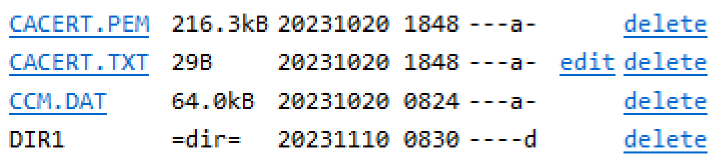
The 4MB FLASH device has an endurance of 100k writes, on a per-byte basis. The FAT directory area is written on every file write and this thus gets the most wear. Care must therefore be exercised when writing code which does file write operations. There is no limit on reading operations since no "last-accessed" timestamp gets written - FAT12/FAT16 file systems have only the "last-modified" timestamp.

Only "8.3" filenames (names in the format xxxxxxxx.xxx) are supported e.g.

filename.txt
file1.txt
f.a
but not filename2.txt3 because >8 or >3 in the filename and the extension, respectively, are not allowed. Wiki article: https://en.wikipedia.org/wiki/8.3_filename

However, nothing stops a USB Host (e.g. Windows) creating whatever filenames and directory structures it is able to according to its own operating system limitations. Within the FAT12 drive it can create more or less anything. If the Host creates a non-8.3 filename, it will be visible to user code under its 8.3 alternate filename which the Host must create, by convention.

All files accessible to user code must be in the **root** of the drive. Subdirectories (folders) are not supported. If a USB Host creates a subdirectory, this will not be accessible to user code although its name will be visible to the KDE_get_file_list() and KDE_get_file_properties() functions. A subdirectory in the root will also be visible in the HTTP Server Files page - example:



```
CACERT.PEM  216.3kB 20231020 1848 ---a-      delete
CACERT.TXT  29B     20231020 1848 ---a- edit delete
CCM.DAT     64.0kB  20231020 0824 ---a-      delete
DIR1        =dir=   20231110 0830 ----d      delete
```

Host Operating System Caching issues

It is important to realise that the USB Host sees the 2MB of FLASH as no more than a number of 512 byte sectors representing a FAT12 removable storage device, which it owns entirely. It can perform whatever operations it wants to in there, without regard to whether the KDE485

internal software understands it. The KDE485 system software is viewing this 2MB FLASH block from the other side and uses the FatFS embedded filesystem module
http://elm-chan.org/fsw/ff/00index_e.html
to interpret the data as a FAT12 filesystem.

Looking at this in the opposite direction (KDE485 to USB Host), files created or modified by the KDE485 do not immediately become visible to the USB Host. This is due to Host operating system architecture; for example Windows assumes nobody else is changing the data on a removable drive, and almost never checks for changes. It checks if it detects that the media has been removed or mounted. See KDE_file_usb_eject() and KDE_file_usb_insert() functions on how to make KDE485 file changes visible to the USB Host.

Files created or modified by the USB Host should become visible to KDE485 code immediately, assuming the data is actually written and the directory updated. This does not always happen because the OS may not flush the data to the USB drive. This was particularly true for older versions of Windows (before winXP).

Filenames are not case-sensitive. All filenames created by user applications are converted to uppercase.

The FLASH device used for the filesystem implements a per-sector pre-read on writes and performs the write only if the data is different. This increases the write speed by around 50x if the data has not changed. It can sometimes appear confusing in that a file writes extremely fast; this is because is has not been changed, or only a few bytes were changed! This is sometimes visible with KDE485 firmware updates or user code loads (firmware.dat).

File System Special Files

The following files have special significance:

BOOT.TXT	- read-only data
CONFIG.INI	- configuration file
FAVICON.ICO	- optional alternative favicon for HTTP server
CACERT.PEM	- file holding remote HTTPS server's certificate(s)

BOOT.TXT

This file is generated at each power-up and contains the following values:

```
Boot_time: 2022-04-17 18:34:32       ; ISO 8601 defined YYYY-MM-DD hh:mm:ss
Firmware: 1.1                        ; version of KDE485 factory code
Build: factory                       ; running code: factory or customer
Appname: appname_1.1                 ; string extracted from appname.ini during build
S/N: 87654321                        ; KDE485 serial number
CPU ID: 260054536278190451363847     ; unique CPU ID string (12 hex byte values)
Options: ARINC429 SPI_RAM            ; presence of these two factory options
                                      (the SPI_GPS option is not shown in boot.txt)
IP: 192.168.4.78                     ; current IP (dhcp or statically allocated)
IP mode: D                           ; ‘D’ for DHCP
                                       ‘S’ for statically allocated IP
                                       ‘F’ for static IP used due to DHCP failure
```

Last two IP parameters above are set to empty at KDE startup and are filled-in once the values are known. With DHCP, these are filled-in some tens of seconds after power-up.

The IP is listed so that one can discover the KDE's IP, e.g. for accessing its HTTP server.

There is an issue caused by operating system caching of removable drives: because boot.txt is generated at power-up and then can have items added some seconds later (generally, when an IP is allocated, its value is filled in) the update may not be visible to the USB host immediately. If you want to see the latest boot.txt updates, there are two solutions:

1) Do not open boot.txt (e.g. in Windows Explorer) until the KDE485 is likely to have obtained the IP, which can take tens of seconds depending on the system.

2) To force the USB host to update its cache, the USB drive can be re-mounted by holding down the KDE485 button for more than 2 seconds. A 10-second press with the hex switch set to other than F reboots the KDE485 and this will have the same effect.

CONFIG.INI

This contains KDE485 configuration data and other values. This file is created during factory initialisation and can be edited at any time, via USB, via the HTTP server, or with internal user software.

It can be also used by user code to set and get non-volatile option values - as long as the option names do not conflict with config options used by the KDE485 factory code.

It contains values in the format

name=value

If at KDE485 startup the "config=1" value is missing, any existing config.ini file is deleted and a new one created with the defaults below.

The values in config.ini are read mostly only at startup but some can be read during operation.

The following values are factory defaults:

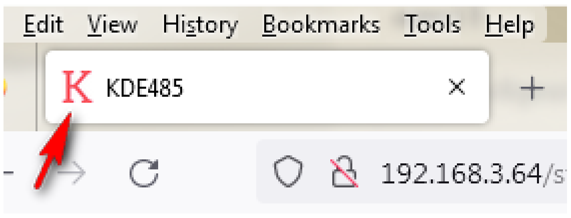
```
config=1                        ; used to detect a valid config.ini file
led_comms=1                     ; 1 = LEDs indicating data flow (port 0-4)

port1=9600,8,n,1                ; port 1 config
port2=9600,8,n,1                ; port 2 config
port3=9600,8,n,1                ; port 3 config
port4=9600,8,n,1                ; port 4 config
p3drv=2                         ; port 3 set for 2 wire 485 and auto driver enable
p4drv=1                         ; port 4 set for RS422 and driver always enabled
p34slew=0                       ; port 3+4 slew rate is low (up to 115200 baud)
p34delay=10,2                   ; port 3+4 driver enable to data delay (µs,slow,fast)

copyser1=0                      ; serial-serial copy task 1 disabled
copyser2=0                      ; serial-serial copy task 2 disabled

eth=1                           ; enable Ethernet

dhcp=1                          ; 1 = obtain IP via DHCP; 0 = disable DHCP
dhcp_retries=10                 ; number of retries
ip_static=169.254.75.75         ; IP allocated if dhcp=0 or if DHCP fails
ip_mask=255.255.255.0           ; used with static IP
ip_gateway=169.254.75.75        ; used with static IP (see also here)
ip_dns_server=8.8.8.8           ; used with static IP
ntp_server=time.windows.com ; 40.119.148.38
ntp_max_diff=10                 ; max jump, in seconds (always +ve value)
ntp_jump_first=1                ; 1 allows unlimited jump on first update
ntp_resync=24                   ; interval (hours) between checks

gps_rtc=0                       ; GPS for position data and RTC updating (-1=SPI-GPS)

healthcheck=0;                  ; 1 enables healthcheck RTOS task
healthcheck_url=https://hc-ping.com/<insert your URL>
healthcheck_interval=60         ; interval (seconds) between checks

dac_init=0,2048                 ; DAC outputs at power-up (2048=halfway)

kal_target=kksystems.com        ; target for keepalive function
kal_interval=60                 ; interval (seconds) between pings

dropbox=0;                      ; 1 enables dropbox RTOS task
dropbox_access_token=<insert your token>
dropbox_app_key=
dropbox_app_secret=
cert_update=0                ; set to 1 to enable automatic update of cacert.pem
cert_update_url=https://curl.se/ca/cacert.pem
cert_update_interval=24      ; check interval in hours

http_svr=1                   ; HTTP server 1=enabled 0=disabled
http_svr_name=kde485         ; login username
http_svr_pwd=12345           ; login password
http_svr_port=80             ; port (optional e.g. 8080, default=80)
http_svr_timeout=3600        ; auto logout in secs (0 disables auto logout)
http_svr_client=0.0.0.0      ; if not 0.0.0.0, used together with name+pwd

debug_usb=0                  ; 1 enables debugs with debug_thread_* functions
debug_https=0                ; 1 enables debugs from https/tls crypto operations
debug_tls=0                  ; 1/2/3 = MbedTLS internal debug level (requires
                               debug_usb=1 also)
https_verify=2               ; 0/1/2 = MbedTLS client-server X509 verification
                               0=not used  1=optional  2=mandatory

sensor_read=0                ; enable RTOS task (0 if analog option not installed)
sensor_type=1                ; 1=PT100 4-wire
sensor_adc_sample            ; ADC sampling rate (1-8 sps)
sensor_wait=1000             ; inter-reading gap in ms
sensor_2w=0.0                ; RTD 2 wire mode total wire resistance in ohms

ethser=0                     ; 1=enable TCP to Serial bridge

app=1                        ; 1=enable default RTOS "APP" task
```

The following values are optional:

```
eth_mac=007fe3d2f401         ; override the ethernet MAC (bit 0 of byte 1 = 0!!)
eth_mtu=1500                 ; override the MTU; max 1500
eth_multi=1                  ; accept incoming non-ARP multicast packets
cjc_corr=2.5                 ; overrides CJC offset (default 2.0°C)
```

FAVICON.ICO

This is optional alternative favicon for HTTP server. A favicon appears in the browser tab



Favicons are in theory optional but some browsers behave strangely without one; they constantly poll the server looking for one. The KDE485 thus transmits a properly formed favicon to the browser.

This file can be used to replace the default favicon. Nowadays, a favicon can have a much higher resolution than the original 16x16, and gzip is therefore preferred. To generate your own favicon, here is one process:

Generate a favicon with photoshop (etc) or https://favicon.io/favicon-generator
Convert the file to .ico with https://convertio.co
GZIP compress it with https://gzip.swimburger.net

CACERT.PEM

This file is a bundle of CA certificates that the HTTPS Ciient function uses to verify that the server is really the correct site you're talking to (when it presents its certificate in the SSL handshake). The bundle should contain the certificates for the CAs you trust. This bundle is sometimes referred to as the "CA certificate store". This file can be obtained from https://curl.se/docs/caextract.html

If you are connecting to a single (private) server then this file will contain the single self-signed certificate for that server.

If the KDE485 has external internet access, this file can be automatically updated - see here.

FIRMWARE.DAT

The user application is loaded by writing a binary file, firmware.dat, to the 2MB filespace and rebooting the KDE485.

The file system needs to have sufficient free space to hold this file. How this is assessed depends on the context. If the file is being written from Windows, there needs to be sufficient free space before starting. If the file is being written from the HTTP server, any file of the same name is deleted first.

File Timestamps

Timestamps on files created by internal KDE485 software are taken from the RTC. This includes files created via the HTTP Server.

Timestamps on files created by an external USB Host are set by the USB Host.

Filesystem API and Thread-Safety

The KDE_* functions below automatically perform file system mount, unmount, file open, close, etc so they can be used standalone. Except where stated, they are mutex-protected and are thread-safe.

The lower level (FatFS-provided) file functions (e.g. f_read() etc) are not mutex-protected; see the Download() function in KDE_http_server.c for an example of how to use them. These functions are rarely needed; in rare cases (e.g. a 1MB file) f_read() can be a lot faster than KDE_file_read(). In general, file access from multiple RTOS tasks, using the FatFS functions, must be avoided.

```
bool KDE_file_write ( char *filename, uint8_t *data, uint32_t length );
```

Writes to a file
If it does not exist it is created, otherwise it is overwritten
Returns true if successful

data	data to write to the file
length	length of data to write

```
bool KDE_file_append ( char *filename, uint8_t *data, uint32_t length );
```

Appends data to a file
If it does not exist it is created
Returns true if successful

data	data to append to the file
length	length of data to append

```
bool KDE_file_delete ( char *filename  );
```

Delete a file
Returns true if successful

```
bool KDE_file_rename ( char *filename_old, char *filename_new );
```

Rename a file
Returns true if successful

```
bool KDE_file_read ( char *filename, uint32_t offset, uint32_t max_length, uint8_t
*data,  uint32_t *length_read );
```

Read data from the named file
Returns true if file exists

offset	starting offset to read from (in bytes)
max_length	maximum number of bytes to read
data	buffer to receive data
length_read	actual number of bytes read

```
bool KDE_get_filespace ( uint32_t* size, uint32_t* space );
```

Returns true if no error

size	file system size in bytes
space	file system free space in bytes

```
uint32_t KDE_get_file_list ( char *flist, uint32_t maxcount );
```

Scans root directory, fills caller supplied array flist with the 8.3 names, sizes and attributes, up to maxcount of files. Each entry takes up 17 bytes so flist needs to be big enough for maxcount*17.
Returns # of files actually found.
The array is not initialised so you have to use the returned value to determine how many filenames were actually loaded.
Any filename shorter than 12 bytes (the 12 includes the dot) has a 0x00 appended.
Directories are found also.
Each entry in flist is 17 bytes:

0..11	filename
12..15	filesize (uint32_t, LSB in byte 12)
16	attribute byte:

AM_RDO	0x01	Read only
AM_HID	0x02	Hidden
AM_SYS	0x04	System
AM_DIR	0x10	Directory

AM_ARC	0x20	Archive

Example:

```
#define MAXFILES 100             // make room for up to 100 files
static uint8_t filelist[MAXFILES*17];
uint32_t fcount=0;
uint16_t idx=0;
memset(filelist,0,1700);
fcount = KDE_get_file_list( (char*) filelist, MAXFILES);
debug_thread_printf("files found=%d", fcount);
while ( filelist[idx] != 0 )
{
   uint32_t fsize4 = filelist[idx+12] | (filelist[idx+13]<<8) |
    (filelist[idx+14]<<16) | (filelist[idx+15]<<24);
   uint8_t fattrib = ' ';
   if ( (filelist[idx+16] & AM_DIR ) != 0 )
   {
      fattrib='D';
   }
   debug_thread_printf("%12.12s size=%7ld %c",&filelist[idx],fsize4,fattrib);
   idx+=17;
}
```

`bool` **`KDE_get_file_properties`** `( char *name, FILINFO *fno );`

Return properties of a file, or of a directory, in the root directory. If file exists, returns true and fills in structure fno, otherwise returns false.

Example:

```
FILINFO fno;
KDE_get_file_properties ( "file123.txt", &fno );
debug_thread_printf("name=%s, size=%ld",fno.fname,fno.fsize);
and FILINFO is predefined as:
typedef struct
{
   DWORD     fsize;             // File size
   WORD      fdate;             // Modified date
   WORD      ftime;             // Modified time
   BYTE      fattrib;           // File attribute
   TCHAR     fname[13];         // File name
} FILINFO;
```

Configuration Data Storage (config.ini)

The following two functions are for convenient storage of configuration parameters in text files, used in **config.ini** and somewhat similar to the Windows .ini system. Each entry has the simple format

name=value

A semicolon ; causes anything after it, up to the end of the line, to be ignored. This is intended for comments.

Example:

```
KDE_file_set_config_value ( "Timeout1", "60", "config.ini" );
```

will produce this line in config.ini:

Timeout1=60

It is ok to write a comment at this stage too:

```
KDE_file_set_config_value ( "Timeout1", "60 ; timeout #1", "config.ini" );
```

will produce this line in config.ini:

Timeout1=60 ; timeout #1

Note that KDE_file_set_config_value() generates a temporary file KDETEMP.TMP whenever it is rewriting an already existing file. This file is not erased and can be kept as a backup of the previous config.

Maximum line length is 200. Maximum name length is 30.

Files created with KDE_file_set_config_value() start with the line

%KDE!filename

where "filename" is the name of the file. However, neither of the two functions checks for this string afterwards.

In most scenarios, config files will be prepared on a PC and will be loaded to the KDE485 via USB or HTTP. As the above examples show, they can also be created from within the KDE485.

An alternative way to store config data is to use the 512k FLASH area; see the functions LF_write() and LF_read() which operate on 512 byte pages.

```
bool KDE_file_set_config_value ( char *name, char *value, char *filename );
```

Sets a configuration name=value in the given file. This will replace an existing value for that name if present or append the name=value to the end of the file. Remounts the USB mass storage device in order to force Windows (or other OS) to update its cached data to notice that the file has been changed. This is important to prevent the filesystem from being corrupted by the host OS using cached FAT/directory information.

Returns true if no error

name	name of config entry
value	value of config entry
filename	filename to write to (e.g. "config.ini")

```
bool KDE_file_get_config_value (char *name, char *value, char *filename, uint16_t
max_length );
```

Reads the value associated with the given name from the given config file.
Does not return comments. See text above regarding where spaces are allowed.
The execution time is approx 130µs per value in the file, until a name match is found.

Returns true if no error

name	name of config entry
value	where to write the value of the config entry if found
filename	filename to read from (e.g. "config.ini")
max_length	maximum length in chars to write to value

Config file permitted characters

The following applies to the two functions above.

Everything is **case sensitive**.

Non-printing characters (everything in the range 0x01 to 0xFF) **are** allowed within either field, except space (see below), tab, CR, LF, = ; some of which have special functions as described above. Be aware that the representation of characters outside the basic US ASCII set (A-Z a-z 0-9 etc) is nonstandard and editing tools often use multi-byte sequences.

Spaces or tabs are **not** allowed within the **name**, and the = must follow the name immediately. KDE_file_set_config_value() trims the name accordingly before writing out the file.

Spaces and tabs **are** allowed **within** the **value** field. KDE_file_set_config_value strips off leading spaces or tabs (those immediately following the =). Similarly, KDE_file_get_config_value strips off any trailing spaces.

A null **value** is allowed e.g.

```
KDE_file_set_config_value ( "Timeout1", "", "config.ini" );
```

and this creates

Timeout1=

The UTF BOM (byte order mark 0xEF 0xBB 0xBF) is searched for at the start of a config file and is discarded if found. Some text editors insert this at the start of a file.

Line endings are allowed in both CRLF and LF (unix style) but CRLF is preferred. The Edit function in the HTTP Server saves the file with CRLF line endings regardless.

Miscellaneous Filesystem Functions

The following functions are needed because Windows, as the USB Host, has no means of detecting file system changes on removable mass storage media, other than discovering (during its periodic poll of USB storage devices; around 1Hz) that the media has appeared or disappeared.

There is no completely satisfactory solution for the scenario where the KDE is writing or modifying files and the USB Host may be trying to do the same!

```
void KDE_file_usb_eject ();
```

This forces removal of the USB block device. It should be used after updating the file system by user application, to force the USB Host to flush its disk cache and detect the changes.

This function has a similar effect as unplugging and replugging the USB cable - except that it does not affect the USB COM port.

This function is also automatically called when one of the **config** filesystem modifying functions is called (e.g. KDE_file_set_config_value). It is also called for KDE_file_delete(), KDE_file_rename() and KDE_file_format(). There is a 5 second delay from the last of these filesystem modifications before KDE_file_usb_eject() is called. However, it is not called for the normal file write functions because doing so could make the filesystem continuously inaccessible to the USB Host if e.g. a file was appended to every few seconds. In such a scenario, if there is a possibility that a USB Host may be connected at the time **and** it is expecting to see a valid file, you need to implement your own periodic KDE_file_usb_eject() call.

```
void KDE_file_usb_insert ();
```

This re-inserts the USB block device. If this function is called, it should be called at least 5 seconds after KDE_file_usb_eject() to allow the host system to notice that the USB device has been removed and force it to clear its cached view of the file system.

In the **Windows** USB host context, this function is not necessary. KDE_file_usb_eject() is sufficient by itself, and Windows should rediscover the drive after a few seconds. However, it may not do so, and other operating systems may behave differently.

Example:

```
KDE_file_write("file0.txt", "some text", 10);
KDE_file_usb_eject ();
osDelay(5000);           // *** these are not necessary for a Windows host
KDE_file_usb_insert ();  // ***
```

```
void KDE_set_label (char *name);
```

Sets the drive volume label.
Note that various characters are not allowed in drive labels; this is host OS dependent.
Also note that changes may not appear for a long time, due to OS caching the label.

```
void KDE_get_label (char *name);
```

This returns the current drive label, null-terminated.

The following two functions are used to prevent the USB Host accessing the file system while the user application is modifying it. The KDE_ functions provided already do this protection.

```
KDE_file_system_mount ();
KDE_file_system_unmount ();
```

This example of the KDE_file_delete function shows the usage:

```
bool KDE_file_delete ( char *filename )
{
   KDE_file_system_mount();
   // make the file writable if it isn't already
   FRESULT rc = f_chmod(filename, 0, AM_RDO|AM_SYS|AM_HID);
   if (rc == FR_OK)
   {
      rc = f_unlink(filename);
   }
   KDE_file_system_unmount();
   return rc;
}
```

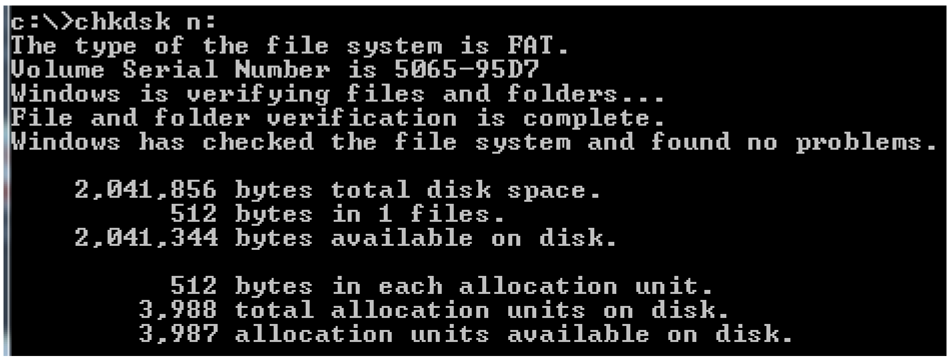
The function f_chmod() is a part of the open source FatFs generic FAT filesystem http://elm-chan.org/fsw/ff/00index_e.html which is used in the KDE485. The above example shows how to create your own wrapper for any of the FatFS functions.

```
bool KDE_file_system_format (void);
```

Formats the file system. This is the equivalent of a "quick format" and takes around 1 second. It does not "wipe" the file system. Returns true if successful.

It is not normally needed since the KDE file system is factory formatted, and can be reformatted anytime via USB using the Windows format command.

However, being a FAT volume, the KDE file system is susceptible to lost clusters in the same way as any other, if e.g. power was interrupted during a file write, and these can be sorted out with a format, or with chkdsk /f



```
c:\>chkdsk n:
The type of the file system is FAT.
Volume Serial Number is 5065-95D7
Windows is verifying files and folders...
File and folder verification is complete.
Windows has checked the file system and found no problems.

    2,041,856 bytes total disk space.
          512 bytes in 1 files.
    2,041,344 bytes available on disk.

          512 bytes in each allocation unit.
        3,988 total allocation units on disk.
        3,987 allocation units available on disk.
```

```
bool filecrc (char * filename, uint32_t crcinit);
```

This reads the file and checks whether its last four bytes are a valid CRC32. The CRC is appended in little-endian format i.e. if the CRC is 0xaabbccdd then the last byte of the file will be 0xdd.

The value crcinit is normally initialised to 0xffffffff by the caller.

Minimum file size is 5 bytes and there is no upper limit. This function uses a 512 byte buffer on the stack. Returns true if CRC is good, false otherwise (including if file doesn't exist).
Execution time for 1MB: 5-10s.

Linear FLASH data storage area

The linear data storage area is accessible in 512 byte blocks. It can be used for any purpose but is particularly intended for data which is being added to frequently but which, if stored in a file in the FAT file system, would quickly reach the 100k write limit of the file system directory. For example, a data logger could store sequentially received data in RAM and each time it has 512 bytes, write that to the FLASH, incrementing the block # (mod 1024), and thus storing the most recent 512k bytes, and the 100k write limit will not be reached until it has gone around the 512k area 100k times, which translates to approximately 50GB of data written.

The API is in two parts: basic functions which access a single block, and "data logger" functions which implement a higher level interface which ensures wear levelling.

```
bool LF_write ( uint32_t page, uint8_t *data );
```

Writes 512 bytes, into page 0-1023, from buffer data.
Writes only if something on the page has changed from existing FLASH data.
Returns false if page # is out of range.
Execution time ~15ms (if write takes place).

The FLASH device used (Adesto AT45DB321E) has a 100k page write endurance but - as with most FLASH devices - has a further limitation called "adjacent cell disturbance". This imposes a lower limit but it applies only to writes within the same 64k block, and the effect is that data can get modified; the cells do not wear out. There is no practical way to work around this but in the KDE485's application area all this is irrelevant unless one is doing a highly unusual writing activity. It is recommended that pages within this 512k area are written such that the writing is "walking along" the storage area.

```
bool LF_read ( uint32_t page, uint8_t *data );
```

Reads page 0-1023, into buffer data which needs to be 512 or more bytes in size.
Returns false if page # is out of range.
Execution time is 210 µs if buffer is in main RAM ("static"), 310 µs if buffer is on the stack.

```
bool KDE_FLASH_Status (void);
```

Returns FLASH programming status: True=Ready.
Execution time 5 µs.
This function can help with RTOS code optimisation. It enables the detection of when a FLASH programming cycle (~15ms) is running.

Linear FLASH "Data Logging" Functions

This API builds on the LF_write and LF_read functions to deliver an interface for the storage of data in typical data logging applications, which uses the FLASH in a manner which maximises its life.

This provides a way to store a sequence of fixed size records in flash memory which can be read & erased from at the beginning and written & appended to at the end. It uses a system of markers to avoid a frequent rewriting of the same FLASH locations. The API allows sequential access using push/pop and random read access.

`int32_t` **`KDE_LF_open`** `();`

Finds start and end markers, performs integrity checking/fixing and determines length of available data. Stores start page/offset and end page/length in RAM.
Returns number of pages, or -1 in case of failure.

`void KDE_LF_close ();`

This function does nothing; it is provided for compatibility with future implementations.

`Int32_t` **`KDE_LF_push`** `(uint8_t *data, uint32_t length);`

Add a new record at the end of the sequence.

`uint32_t` **`KDE_LF_read`** `(uint8_t *data, uint32_t max_length);`

Read data from the current page and increment the page position.
Returns actual number of bytes copied, or 0 if there is nothing more to read.

`bool` **`KDE_LF_read_seek`** `(uint32_t page offset);`

Set the page position relative to the start of the sequence.
Returns true if the given offset is valid (i.e. less than the total file length). When false is returned, the offset is beyond the end of the file and the (internal) current read position is not altered.
Note that this function only works when the log file system isn't used by other RTOS tasks.
Other tasks may set a different seek position.

`uint32_t` **`KDE_LF_get_page_size`** `();`

Return the amount of data that can be stored inside a record (at least 511 bytes).

`int32_t` **`KDE_LF_get_records`** `();`

Returns the number of records in the sequence. Note that the number of records may change between function calls due to other threads accessing the log file system. Don't use the result of this function to run a loop in order to remove or read all data. When needing to read all data: call KDE_LF_read() until it fails. To remove all data, call KDE_LF_pop() until is fails.

`int32_t` **`KDE_LF_get_read_pos`** `();`

Returns the current page position, or -1 in case of failure.

`bool` **`KDE_LF_pop`** `(uint8_t *data, uint32_t max_length);`

Removes the first page of the data sequence. If *data is not a NULL pointer and max_length is >0 then the data will be read first.
Returns true when a page was removed, or false is there is nothing left in the sequence.

Logging data to RAM: As an alternative for heavy data logging and similar applications, note the **SPI RAM (8MB)** factory option. This provides 8MB of data storage, in 512 byte pages, with high speed (around 1MB/sec), and with no wear-out mechanism.

Power Down Data Save

This feature enables 512 bytes of user data to be saved in a dedicated page in the serial FLASH upon the loss of KDE power. It works for either the 12/24V DC input or, if the KDE is being USB-powered, the 5V USB power. It is implemented using the internal ADC1 A-D converter which checks an internal supply rail every 1ms. If enabled, ADC1 is no longer available to user code.

To achieve a sufficiently fast programming cycle, a special mode of the FLASH device is used which requires the FLASH page to be already erased. Accordingly, **KDE_power_down_save_enable(true)** erases that page, so you lose data saved previously. **KDE_power_down_save_enable(false)** does not affect the stored data, but requires a previous call with (true) to erase the page.

```
bool KDE_power_down_save_enable (bool status);
```

Controls whether the data save feature is enabled: True=enable.
Returns false if FLASH erase failed.
Execution time 20ms (if enabling) 10us otherwise.

```
bool KDE_power_down_save_data_write (uint8_t *data);
```

The 512 byte buffer "data" is copied to a 512 byte RAM buffer in the FLASH chip. This buffer is then programmed into the FLASH at power-down.
Returns false if the FLASH page is not erased, or if the power down data save feature is not enabled.
Execution time 1ms.

```
void KDE_power_down_save_data_read (uint8_t *data);
```

data: a 512 byte buffer.
Execution time 300us.

This function can be called at any time and returns the content of the FLASH page dedicated to this feature. Whether it contains valid data (from the preceeding power-down) can be determined only by an inspection of the data; see e.g. the CRC suggestion below.

In normal operation, it is expected that user code maintains data to be saved in a buffer and periodically call **KDE_power_down_save_data_write(buffer**).

At each startup, user code reads the data with **KDE_power_down_save_data_read()** and if the data is valid (good CRC, etc), utilise it. Then, it calls **KDE_power_down_save_enable(true**) to set up for the next power-down.

If the data save feature is no longer required, call **KDE_power_down_save_enable(false).**

Data Save limitations

There is no integrity checking so you may need to implement a CRC or similar, within the 512-byte buffer.

The data save will fail if there is a FLASH **write** (~15ms) in progress when the power fails. That write will also fail. The data save will also be skipped if a FLASH **read** is in progress, but should be executed 1ms later if the read was brief.

The data save feature is internally disabled for 10 seconds after KDE power-up, to avoid spurious FLASH writes if the power supply is being cycled rapidly.

A reasonably clean power-down is assumed. It is not possible to test such a scheme with every possible kind of power transition or brown-out.

The above functions allocate up to two 512 byte buffers on the stack so you need extra stack allocation for the RTOS task calling this.

Serial Ports

Four serial ports are provided, plus a USB VCP (virtual COM port):

Port	Interface	Baud rate range
P0	USB VCP	N/A
P1	RS232	1200-115200
P2	RS232	1200-115200
P3	2 wire RS485, half duplex	1200-1843200
P4	RS422, or 4 wire RS485	1200-1843200

P2 supports a mode with RTS/CTS hardware handshake. When this mode is enabled, P1 TX/RX signals become RTS/CTS for P2. See below. This mode is accessible only from a user program (not from config.ini). Hardware handshake is very rarely used nowadays.

Note the differing baud rate ranges between ports. This is due to the controlled slew rate drivers used for low EMC on P1 and P2. Also, for EMC, the P3 and P4 max rate of 1843200 drops to 115200 if using KDE_serial_485_slew_rate_control() with a parameter of 1.

At startup, the four ports are initialised with values in config.ini.

Serial port initialisation and baud rate ranges

Two initialisation functions are provided. The first, KDE_serial_set_port_config, is recommended for all new projects. It supports the full capabilities of the hardware.

```
bool KDE_serial_set_port_config ( uint8_t port, SP_CONFIG *config );

struct {
      uint32_t bit_rate;        // actual baud rate value as an integer
      uint8_t bits_per_word;    // 7 or 8
      uint8_t parity;           // 0,1,2 = none,even,odd parity
      uint8_t stop_bits;        // 1,2 = 1,2 stop bits
      uint8_t rx_rts;           // 0,1 = off, automatic (port 2 only)
      uint8_t rx_dtr;           // reserved, set to 0
      uint8_t rx_xon;           // reserved, set to 0
      uint8_t tx_cts;           // 0,1 = off, automatic (port 2 only)
      uint8_t tx_dsr;           // reserved, set to 0
      uint8_t tx_xon;           // reserved, set to 0
      uint8_t tx_enabled;       // 1 = enabled
      uint8_t rx_enabled;       // 1 = enabled
      uint8_t reserved;
} SP_CONFIG;
```

The KDE485 uses a special baud rate generation method which enables unusual baud rates - not only direct submultiples of some crystal frequency - to be generated. The value bit_rate can be any integer in the valid baud rate range, so e.g. 23456 baud is possible. The actual range of the UART is 642 baud to over 2.5mbaud. At 115200 and below the error is < 0.2%, increasing to 0.7% at 1843200.

If rx_rts=1 then P1 TX is disabled and P1 TX becomes an RTS signal for P2. RTS acts as a receive handshake for data arriving on P2.

If tx_cts=1 then P1 RX is disabled and P1 RX becomes a CTS signal for P2. CTS acts as a transmit handshake for data coming out of P2.

Serial Comms Functions

Unless specified, these apply to all five ports i.e. 0-4.

```
bool KDE_serial_transmit(uint8_t port, uint8_t *data, uint16_t length);
```

returns false if there is insufficient space for the given length

```
bool KDE_serial_transmit_UL(uint8_t port, uint8_t *data, int32_t length);
```

similar to above but handles any size block, and loops until all sent, yielding to RTOS appropriately

```
uint16_t KDE_serial_receive(uint8_t port, uint8_t *data, uint16_t maxlength);
```

returns # of bytes read

```
uint16_t KDE_serial_get_ipqcount (uint8_t port);
```

Returns the number of bytes available in the receive buffer for the port (0-4).

```
uint16_t KDE_serial_get_opqcount (uint8_t port);
```

Returns the number of bytes in the output queue for the port (0-4).

The returned value includes any in the UART waiting to go out, but due to hardware limitations a return value of zero means the transmitter may still be shifting out the last byte; this is relevant only with RS485 driver control where the KDE_serial_485_driver_control() function addresses this issue.

```
uint16_t KDE_serial_get_opqspace (uint8_t port);
```

Returns the number of bytes of free space in the output queue for the port (0-4).

```
void KDE_serial_flush_rx (uint8_t port);
```

Clears the input queue for the port (0-4).

```
void KDE_serial_flush_tx (uint8_t port);
```

Clears the output queue for the port (0-4).

```
uint8_t KDE_serial_485_driver_control (uint8_t port, uint8_t mode);
```

This function is for use on ports 3,4 only. It enables fully transparent RS485 driver control on these two ports:
0 - driver off
1 - driver enabled
2 - auto - driver enabled when transmitting, disabled automatically after the end of transmission

Returns the current state of the driver: 0 off; 1 on

Mode=0 stops any data coming out of ports 3,4. It may be used for e.g. 2-wire RS485 bus monitoring.
On port 4, mode=1 is used for RS422 or for a Master on a 4-wire RS485 bus.

Ports 3,4 mode is initialised at power-up (from config.ini - p3drv and p4drv) to 2 and 1 respectively, corresponding to port 3 being 2-wire RS485 and port 4 being generally RS422.

If KDE_serial_set_port_config() is used, KDE_serial_485_driver_control should be called afterwards to override the config.ini settings.

```
void KDE_serial_485_slew_rate_control (uint8_t speed);
```

Sets RS422/485 slew rate (maximum port speed) for ports 3,4.

On ports 1,2 (RS232) the max baud rate is 115k and this is limited by the driver chips (MAX232 or equivalent). On ports 3,4 (RS422/485) the max baud rate is limited by the driver chips (MAX3089 or equivalent) to 10mbps but this is switchable to 115k to reduce cable EMC emissions. Note that all four of the KDE485 UARTs support up to 1843200 baud.

Ports 3,4 are initialised at power-up (from config.ini - p34slew) to 0 (115k max) for lowest EM emissions. Set the speed to 1 for baud rates above 115k.

Example:

The following example will output the string "abcd", at 10ms intervals, from port 4, with the RS485 driver enabled around each 4-byte string:

```
KDE_serial_485_slew_rate_control(1);    // high speed mode
KDE_serial_485_driver_control (4,2);    // automatic RS485 driver mode
while (true)
{
   kfprintf (4, "abcd");
   osDelay(10);
}
```

The value given to KDE_serial_485_slew_rate_control() also sets a wait time from RS485 driver enable to the start of transmitted data, to 10µs or 2µs for the slow and fast mode, respectively. These values are configurable in config.ini with the p34delay=x,y parameter, where x,y are the delays in µs for the slow and fast mode respectively.

The KDE485 has a factory build option where both Port 3 and Port 4 are RS232. In this case, KDE_serial_485_slew_rate_control() has no effect.

The following functions are provided for compatibility with KD485-PROG programs:

```
int setportcfg ( struct sp_t *config );

struct sp_t
{
      int sp_port;              // 1,2,3,4 = p1,p2,p3,p4
      int sp_brate;             // baud rate index 8..20 (see table below)
      int sp_bword;             // 2,3 = 7,8 b/word
      int sp_pty;               // 0,1,2 = none,even,odd parity
      int sp_sbits;             // 0,1 = 1,2 stop bits
      int sp_rxrtshs;           // 0,1 = off, automatic
      int sp_rxdtrhs;           // reserved, set to 0
      int sp_rxxhs;             // reserved, set to 0
      int sp_txctshs;           // 0,1 - off, automatic
      int sp_txdsrhs;           // reserved, set to 0
      int sp_txxhs;             // reserved, set to 0
};
```

sp_brate	baud rate	sp_brate	baud rate
10	1200	15	7200
11	2000	16	9600
12	2400	17	19200
13	3600	18	38400
14	4800	19	57600
		20	115200

```
int ipqcount(int port);
int opqcount(int port);
int opqspace(int port);
int ipqclear(int port);
char kfgetc(int port);          // Blocking call to get a character
int kfputc(int port, char ch);  // Blocking call to put a character
int freadc(int port, char *buffer, int count);  // Blocking call to read count
                                                   characters
int fwritec(int port, char *buffer, int count);  // Blocking call to write count
                                                   characters
int kfprintf(int port, char *buffer, ...);    // Uses a 128 byte buffer on the
                                                   stack and uses vsnprintf();
                                                   use with caution
void drvctrl(int port, int enable_state);
void wait_485(int port, int drvstate);
```

USB Virtual Com Port (VCP - port 0)

This is a bidirectional port, implemented with the USB CDC device profile. There is no baud rate associated with a VCP and any value configured on that port has no effect. The speed is generally very high: 50+ kbytes per second.

A driver, which produces a COM port within the operating system, is available for Windows 7 (see the Utils directory). For later Windows versions, no driver is required.

Port 0 **output** (KDE485 to PC) is supported with **KDE_serial_transmit()** and (primarily) **KDE_serial_get_opqspace().** The latter is normally polled to make sure there is enough room for the data. The data rate over USB is up to 50 kbytes/sec but is otherwise limited by how fast the USB Host is retrieving the data. As with ports 1-4, there is a 1k transmit buffer.

Port 0 **input** (PC to KDE485) is supported with **KDE_serial_receive()** and (primarily) **KDE_serial_get_ipqcount()**. The latter is normally polled to see if any data has arrived. As with ports 1-4, there is a 1k receive buffer. The data rate is similarly high.

Note that the USB VCP implementation has proper flow control in both directions. This means that if you are running a slow application on the VCP on the PC, the KDE485 output rate will be limited by that. And similarly if the application running on the KDE485 is slow in extracting the incoming data.

USB flow control has limitations. One is that if the USB Host has no application connected on the VCP COM port, or the USB port has somehow failed, and the KDE485 is generating port 0 output, the KDE485 may not start up or may run slowly. The following safeguards are thus implemented which lead to discarding of output data:

- if there is no +5V (VBUS) arriving from the USB cable
- if a USB "busy" state persists for >1000ms
- if the USB Host does not have a CDC device profile

However, a comprehensive detection of a "dead USB Host" is not possible to achieve reliably. **Therefore, if not using the USB VCP function, but a USB cable is connected to a USB Host (powered or not), ensure all debug_* items in config.ini are set to 0.**

A further limitation on port 0 usage is if **TLS** (HTTPS) is in use: serial packet size should be below 200 bytes and in half duplex operation the receive device timeout needs to be set to >5 seconds.

The availability of the port 0 VCP function for "**normal" serial data** depends on the value of the **debug_usb** parameter in config.ini. This value controls the output of the functions debug_thread() and debug_thread_printf(). The KDE485 system software emits various debugs and if debug_usb>0 these will come out of the USB VCP, corrupting your port 0 data.

Port-Port Serial Data Copy

This is a bidirectional data transfer application which forms 1 or 2 channels, each transferring duplex data between any two ports. It is equivalent to KD485-ADE Mode 1. If a port is capable of half-duplex (port 3 or 4) then its driver is automatically enabled when transmitting. The baud rates etc can also be different. All five ports 0-4 are supported, with port 0 being the USB VCP (virtual COM port).

This feature is enabled in config.ini with e.g.

```
copyser1=1,3
copyser2=2,0
```

where channel 1 copies data between ports 1 and 2, and channel 2 copies data between ports 2 and 0. To disable a channel, omit the "copyser" line, or specify just one port instead of two e.g.

```
copyser1=0
```

Debug Output

This section deals with generating debug messages from user programs, and from the KDE485 factory software.. Apart from outputting data from any of the four serial ports, two specific debug output channels are provided:

- USB VCP (virtual COM port)
- SWV ITM Data Console, via Cube IDE or ST-LINK Utility (requires a STLINK V3 debugger)

USB VCP Debugging Functions

For outputting debug messages from an RTOS task, where a terminal program connected to a USB VCP (e.g. Teraterm) is running, the functions

void **debug_thread** (char * text);
void **debug_thread_printf** (char *format, ...);

are provided. These accept a string up to 128 bytes long. The 2nd supports the usual printf formatting functions, via snprintf. Each line is transmitted atomically via USB CDC (even if other data is being sent out of port 0) so lines from different RTOS tasks are not broken up. Each line starts with a millisecond counter of time since startup:

```
214837797:   Verifying peer X.509 certificate...
```

The above debug functions are implemented **without** USB flow control and with a short inter-block delay instead (2ms). This is necessary, otherwise if you had a statement like

debug_thread("Task 1 starting");

at the start of an RTOS task, and didn't have an application running on the PC and connected to the COM port, and the port 0 buffer (1k) filled up with output from other tasks, this task would never run past the debug statement because debug_thread() would block. And depending on task priorities this could make the KDE485 run slowly.

The KDE485 can generate data to port 0 faster than USB can send it to almost any PC application. Therefore, the debug functions could incur loss of data if called at a high repetition rate.

Limiting USB VCP debugs to a specific user task

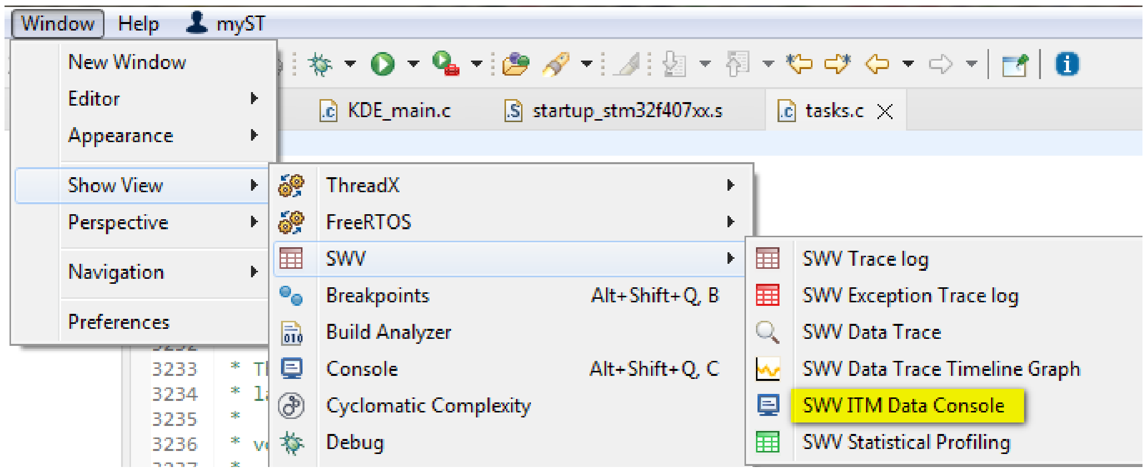
Some KDE485 system software also emits debugs when debug_usb>0. If debug output is required for testing a specific RTOS task **only**, set usb_debug=0 to suppress all these KDE485 system debugs, and the following functions (note the leading underscore) bypass the debug_usb parameter:

```
void _debug_thread (char* text);
void _debug_thread_printf (char *format, ...);
```

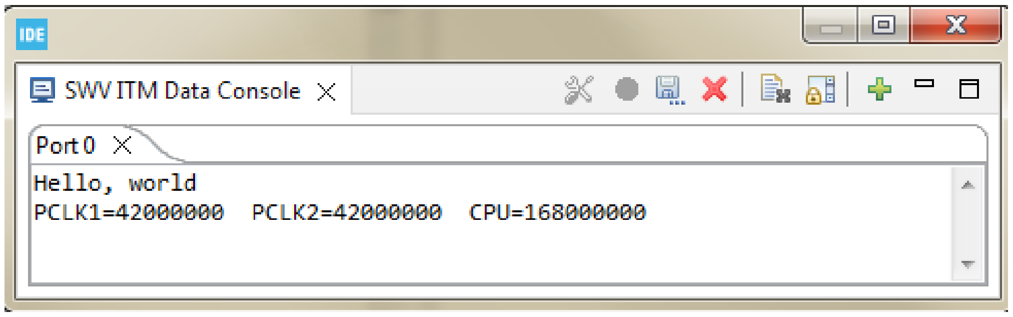
SWV ITM Data Console (Cube IDE, or STM32 ST-LINK Utility)

This is a high speed debug output feature, for minimally intrusive debugging of time-critical code. It is applicable only when running with the STLINK V3 debugger.

In Cube IDE, when running in Debugger mode (program built with F11), the SWV ITM Data Console tab displays the output. With the program running, open the display window with



This feature works best with the SWV ITM tab in Detach mode:



```
Hello, world
PCLK1=42000000   PCLK2=42000000   CPU=168000000
```

Two functions are provided whose output is directed to the SWV ITM Console:

```
void debug_puts (char * string);
```

This efficient function outputs a null-terminated string (no length limit). Source is in KDE_debugprint.c
Example:

```
debug_puts("Hello, world\r\n");
```

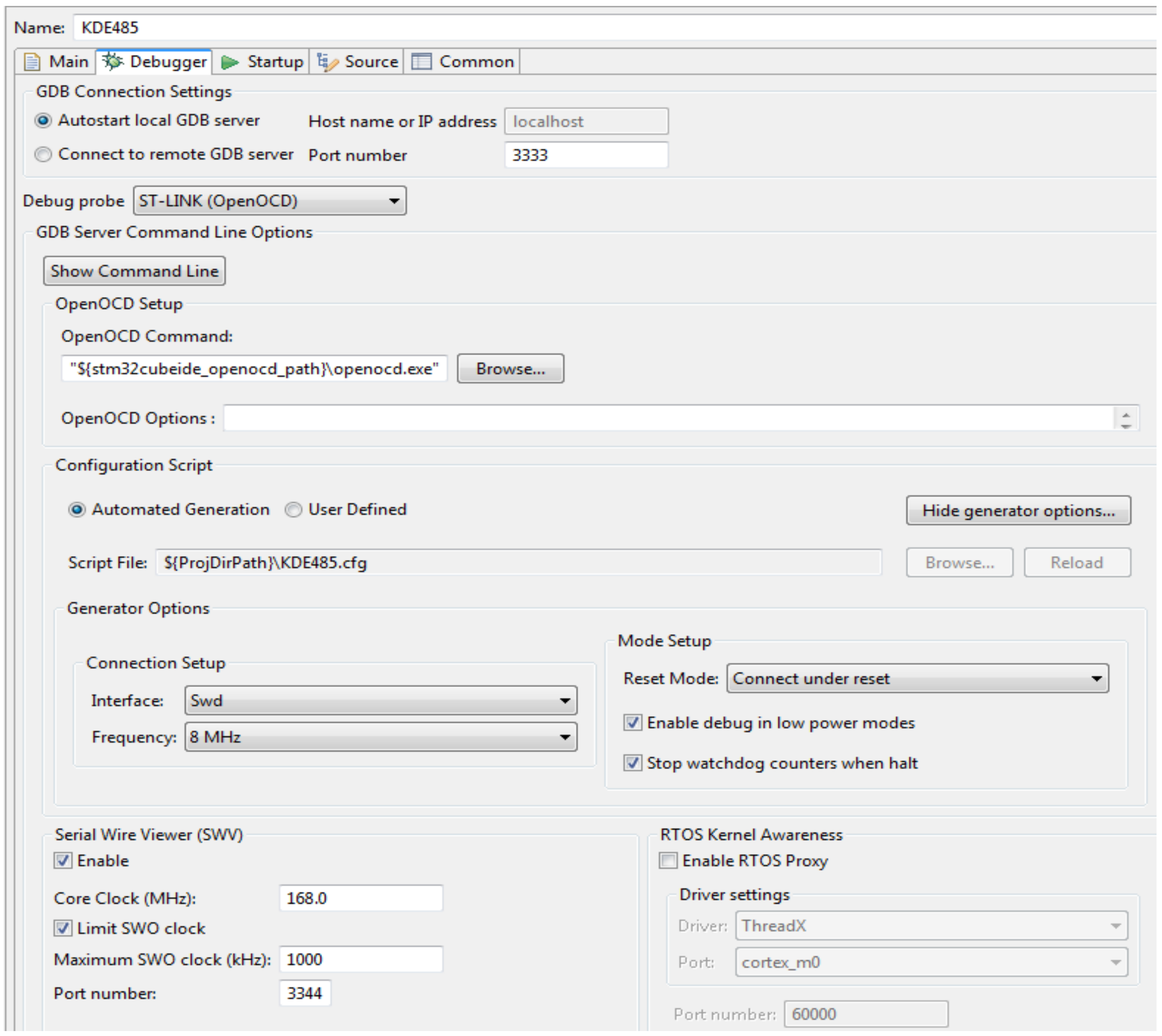
```
int printf (const char* format, ...);
```

This is a standard printf() with all the standard Newlib GCC features. Like the standard printf() it also involves executing a lot of code but is very convenient.
Example:

```
uint32_t ck = 168000000L;
printf("CPU=%ld\r",ck);
```

Note that the output replaces a CR with a CRLF i.e. \r does a \r\n. For a new line, use \n.

In Cube IDE, the SWV ITM Console is configured under Project / Properties / Run / Debug Settings / KDE485 / Edit / Debugger:



Name: KDE485

Main | Debugger | Startup | Source | Common

GDB Connection Settings

- [✓ Autostart local GDB server] Host name or IP address: localhost
- Connect to remote GDB server Port number: 3333

Debug probe: ST-LINK (OpenOCD)

GDB Server Command Line Options

Show Command Line

OpenOCD Setup

OpenOCD Command: "${stm32cubeide_openocd_path}\openocd.exe" Browse...

OpenOCD Options :

Configuration Script

[✓ Automated Generation] User Defined — Hide generator options...

Script File: ${ProjDirPath}\KDE485.cfg — Browse... Reload

Generator Options

Connection Setup

Interface: Swd

Frequency: 8 MHz

Mode Setup

Reset Mode: Connect under reset

[✓] Enable debug in low power modes

[✓] Stop watchdog counters when halt

Serial Wire Viewer (SWV)

[✓] Enable

Core Clock (MHz): 168.0

[✓] Limit SWO clock

Maximum SWO clock (kHz): 1000

Port number: 3344

RTOS Kernel Awareness

[ ] Enable RTOS Proxy

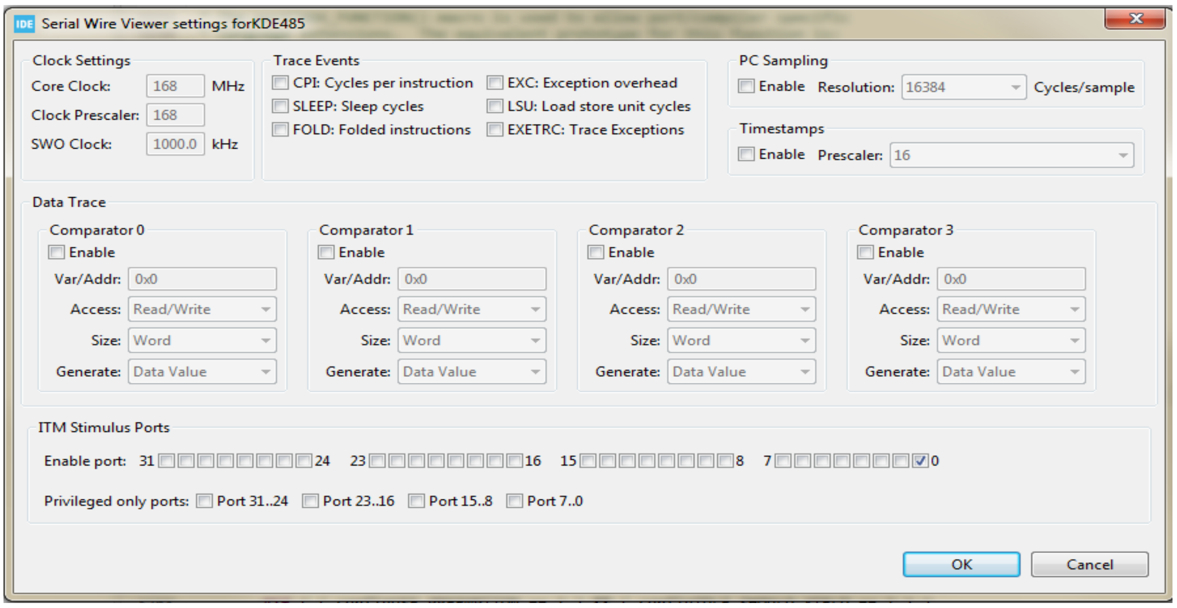
Driver settings

Driver: ThreadX

Port: cortex_m0

Port number: 60000

Then there is the corresponding SWV ITM Data Console configuration:



Serial Wire Viewer settings forKDE485

Clock Settings

Core Clock: 168 MHz

Clock Prescaler: 168

SWO Clock: 1000.0 kHz

Trace Events

- [ ] CPI: Cycles per instruction
- [ ] SLEEP: Sleep cycles
- [ ] FOLD: Folded instructions
- [ ] EXC: Exception overhead
- [ ] LSU: Load store unit cycles
- [ ] EXETRC: Trace Exceptions

PC Sampling

[ ] Enable Resolution: 16384 Cycles/sample

Timestamps

[ ] Enable Prescaler: 16

Data Trace

	Comparator 0	Comparator 1	Comparator 2	Comparator 3
Enable	[ ]	[ ]	[ ]	[ ]
Var/Addr:	0x0	0x0	0x0	0x0
Access:	Read/Write	Read/Write	Read/Write	Read/Write
Size:	Word	Word	Word	Word
Generate:	Data Value	Data Value	Data Value	Data Value

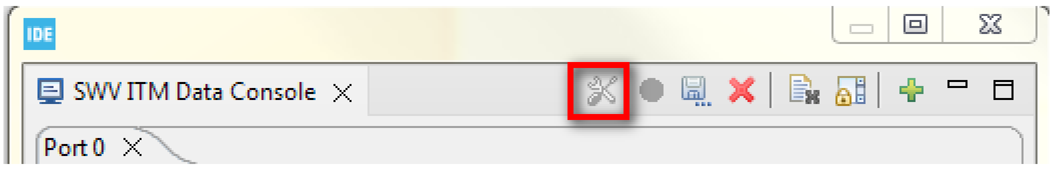
ITM Stimulus Ports

Enable port: 31 [ ][ ][ ][ ][ ][ ][ ][ ] 24 23 [ ][ ][ ][ ][ ][ ][ ][ ] 16 15 [ ][ ][ ][ ][ ][ ][ ][ ] 8 7 [ ][ ][ ][ ][ ][ ][ ][✓] 0

Privileged only ports: [ ] Port 31..24 [ ] Port 23..16 [ ] Port 15..8 [ ] Port 7..0

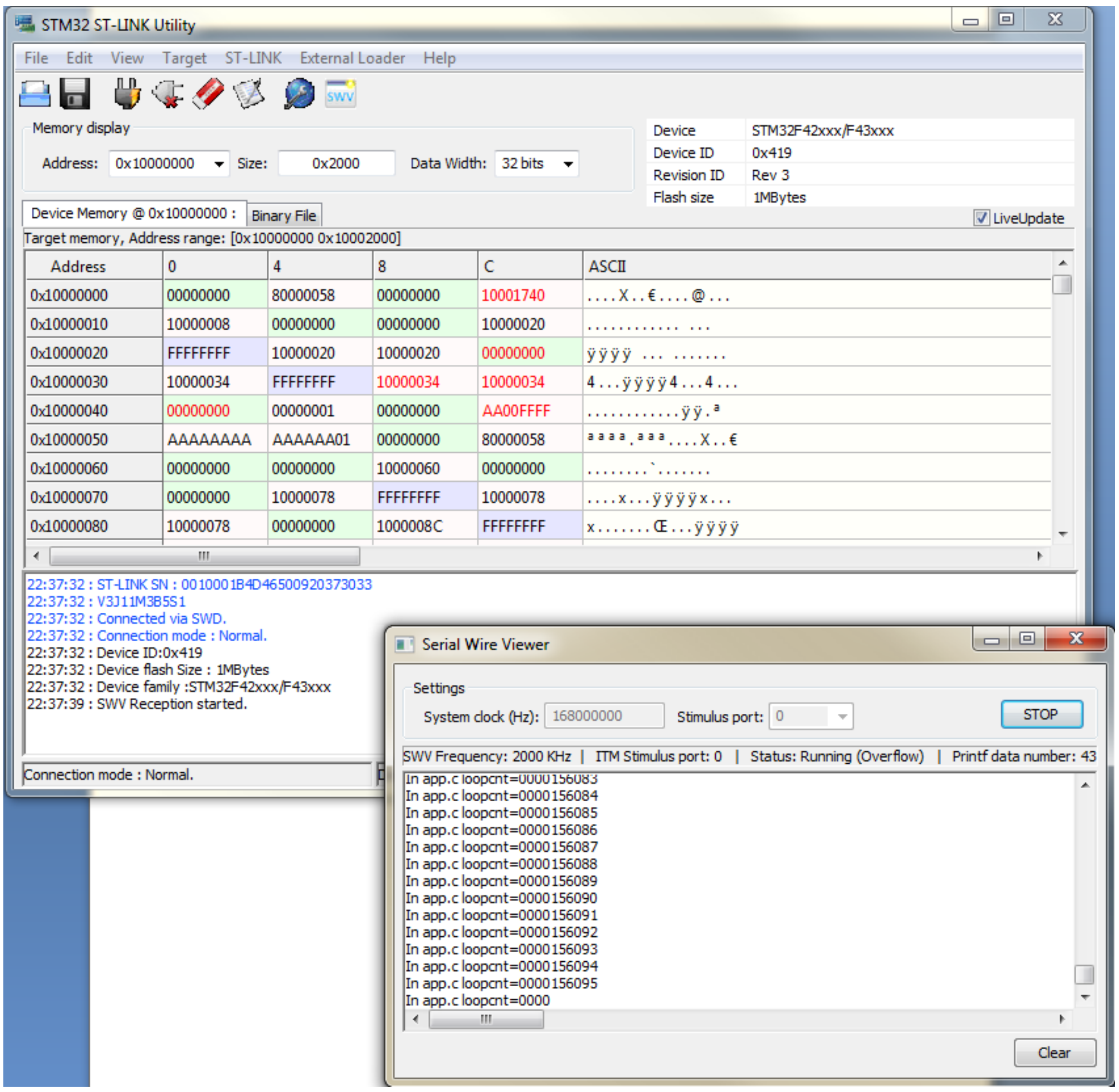
OK Cancel

The above configuration is here



After any configuration change in the the above two screens, Cube IDE must be restarted. It is also advisable to power cycle the debugger (by unplugging its USB cable).

Another way to get the SWV ITM debugs is to use the separate program **STM32 ST-LINK Utility**. This also uses the STLINK V3 debugger. This is a free download:
https://www.st.com/en/development-tools/stsw-link004.html
Here is another URL:
https://www.kksystems.com/customer_area/STM32 ST-LINK Utility v4.6.0.zip



Address: 0x10000000 Size: 0x2000 Data Width: 32 bits

Device	STM32F42xxx/F43xxx
Device ID	0x419
Revision ID	Rev 3
Flash size	1MBytes

Target memory, Address range: [0x10000000 0x10002000]

Address	0	4	8	C	ASCII
0x10000000	00000000	80000058	00000000	10001740	....X..€....@...
0x10000010	10000008	00000000	00000000	10000020	....
0x10000020	FFFFFFFF	10000020	10000020	00000000	ÿÿÿÿ ... .
0x10000030	10000034	FFFFFFFF	10000034	10000034	4...ÿÿÿÿ4...4...
0x10000040	00000000	00000001	00000000	AA00FFFF	............ÿÿ.ª
0x10000050	AAAAAAAA	AAAAAA01	00000000	80000058	ªªªª.ªªª....X..€
0x10000060	00000000	00000000	10000060	00000000	........`.......
0x10000070	00000000	10000078	FFFFFFFF	10000078	....x...ÿÿÿÿx...
0x10000080	10000078	00000000	1000008C	FFFFFFFF	x.......Œ...ÿÿÿÿ

```
22:37:32 : ST-LINK SN : 0010001B4D46500920373033
22:37:32 : V3J11M3B5S1
22:37:32 : Connected via SWD.
22:37:32 : Connection mode : Normal.
22:37:32 : Device ID:0x419
22:37:32 : Device flash Size : 1MBytes
22:37:32 : Device family :STM32F42xxx/F43xxx
22:37:39 : SWV Reception started.
```

Connection mode : Normal.

System clock (Hz): 168000000 Stimulus port: 0

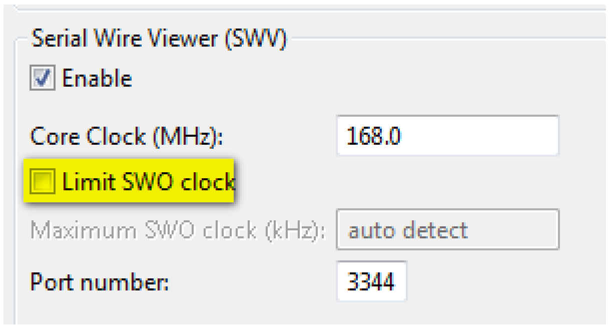
SWV Frequency: 2000 KHz | ITM Stimulus port: 0 | Status: Running (Overflow) | Printf data number: 43

```
In app.c loopcnt=0000156083
In app.c loopcnt=0000156084
In app.c loopcnt=0000156085
In app.c loopcnt=0000156086
In app.c loopcnt=0000156087
In app.c loopcnt=0000156088
In app.c loopcnt=0000156089
In app.c loopcnt=0000156090
In app.c loopcnt=0000156091
In app.c loopcnt=0000156092
In app.c loopcnt=0000156093
In app.c loopcnt=0000156094
In app.c loopcnt=0000156095
In app.c loopcnt=0000
```

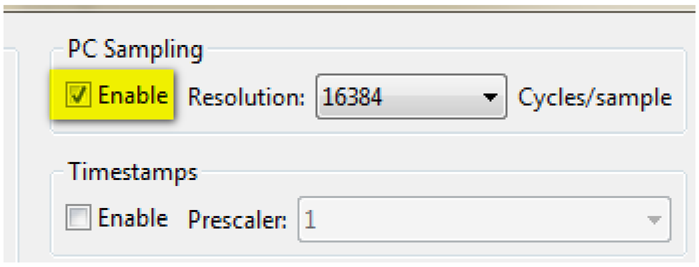
SWV Statistical Profiling (Cube IDE)

This is a powerful feature for finding out where your program spends most of CPU time. In nearly all KDE485 applications, which correctly yield to the RTOS when there is nothing to do, >50% of the time will be spent waiting in the RTOS Idle task. But a highly computationally intensive application may produce a different result.

In this function, high speed data is being collected so the configuration is slightly different:

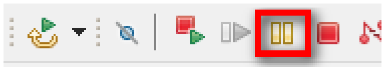


Serial Wire Viewer (SWV)
[✓] Enable
Core Clock (MHz): 168.0
[ ] Limit SWO clock
Maximum SWO clock (kHz): auto detect
Port number: 3344

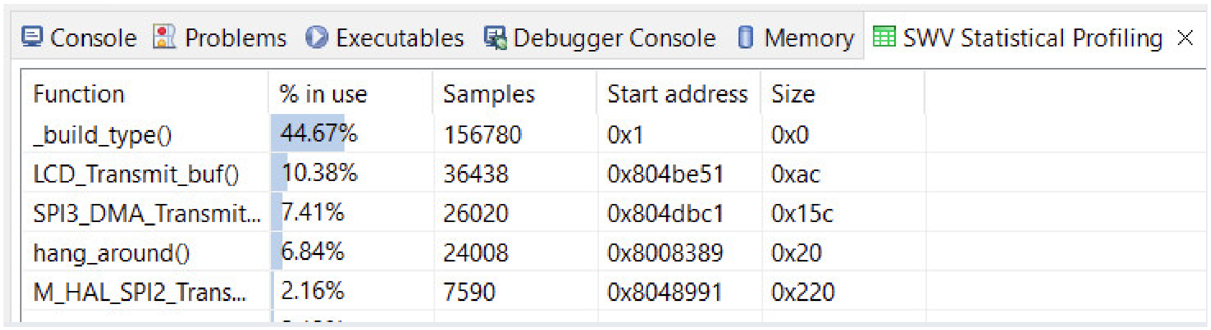


PC Sampling
[✓] Enable Resolution: 16384 Cycles/sample
Timestamps
[ ] Enable Prescaler: 1

Build the program for debugging with F11 and let it run for some minutes. Then pause it with this button



and the Statistical Profiling console should show something like this:



Function	% in use	Samples	Start address	Size
_build_type()	44.67%	156780	0x1	0x0
LCD_Transmit_buf()	10.38%	36438	0x804be51	0xac
SPI3_DMA_Transmit...	7.41%	26020	0x804dbc1	0x15c
hang_around()	6.84%	24008	0x8008389	0x20
M_HAL_SPI2_Trans...	2.16%	7590	0x8048991	0x220

In the above, the _build_type() function is merely the RTOS Idle Task, with a meaningless start address. The other tasks are clickable to view their source code.

Cube IDE is not able to run for very long in this mode, especially with the STLINK debugger. But it does not need to, to accumulate sufficient data. For more serious analysis there are debuggers from e.g. Segger which cost €1000+.

Checksum, XOR, CRC and Random Functions

These functions are provided to support various protocols.

The results in the following examples are based on the 9-byte string

```
char crctest[] = { "123456789" };
```

A simple checksum. Result: 0xdd

```
uint8_t sumbuf ( char *buf, uint16_t buflen );
```

A XOR checksum. Min length = 2 bytes. Result: 0x31

```
uint8_t xorbuf ( char *buf, uint16_t buflen );
```

Two CRC functions are provided. The first requires no initialisation but has fewer algorithm options and is slower. The second requires a table to be initialised (only once after power-up) but offers more options for the CRC algorithm and is much faster. In most applications the speed is a non-issue.

CRC-16 #1

Generic CRC-16. Uses polynomial of 0x8005. The seed of 0 or -1 determines whether the computation will be CRC16-ARC or CRC16-Modbus.

```
uint16_t crc16_1 ( char *buf, uint16_t buflen, uint16_t seed );
```

CRC-16 #2

Table initialisation. This precomputes a 512-byte internal table and specifies some of the algorithm properties.

```
void crc16_2_init ( uint16_t crc_poly, bool crc_dir );
```

Commonly used crc_poly values are

```
#define      CRC_CCITT          0x1021
#define      CRC_16             0x8005
#define      CRC_Rev_CCITT      0x8408
#define      CRC_Rev_16         0xA001
```

The crc_dir value selects one of two bit orders for the computation.

CRC generation.

```
uint16_t crc16_2 ( char *buf, uint16_t buflen, uint16_t seed );
```

CRC-32

```
void crc32 ( uint8_t input_byte, uint32_t *crcvalue );
```

This is the most common CRC-32 variant. Polynomial is 0x04c11db7.

This version accepts one byte at a time, and maintains the CRC in crcvalue. This makes it suitable for calculating a CRC across a number of data blocks.
crcvalue is normally initialised to 0xffffffff (-1) by the caller, and it holds the accumulated CRC as you go along. The final value is the common ISO/HDLC version used on Ethernet, or it can be inverted for JAMCRC. Execution time is 400kbytes/sec.

The following table shows the output of all the functions above, for various parameters and with the buffer containing the aforementioned 9 bytes

```
sumbuf                                    =  0xdd
xorbuf                                    =  0x31

crc16_1, seed= 0                          =  0xbb3d *
crc16_1, seed=-1                          =  0x4b37 ***
crc16_2, CRC_CCITT, seed=0, dir=true      =  0xb955
crc16_2, CRC_CCITT, seed=0, dir=false     =  0x0c73
crc16_2, CRC_CCITT, seed=-1, dir=true     =  0x6eb4
crc16_2, CRC_CCITT, seed=-1, dir=false    =  0x1dba
crc16_2, CRC_Rev_CCITT, seed=0, dir=true  =  0x60c7
crc16_2, CRC_Rev_CCITT, seed=0, dir=false =  0x2189 **
crc16_2, CRC_Rev_CCITT, seed=-1, dir=true =  0x082d
crc16_2, CRC_Rev_CCITT, seed=-1, dir=false=  0x6f91
crc16_2, CRC_16, seed=0, dir=true         =  0x0265
crc16_2, CRC_16, seed=0, dir=false        =  0xafad
crc16_2, CRC_16, seed=-1, dir=true        =  0x82a7
crc16_2, CRC_16, seed=-1, dir=false       =  0x3d7b
crc16_2, CRC_Rev_16, seed=0, dir=true     =  0xa091
crc16_2, CRC_Rev_16, seed=0, dir=false    =  0xbb3d
crc16_2, CRC_Rev_16, seed=-1, dir=true    =  0xe0db
crc16_2, CRC_Rev_16, seed=-1, dir=false   =  0x4b37 ***

crc32, CRC-32/ISO/HDLC, seed=-1,           = 0xcbf43926
crc32, CRC-32/JAMCRC, seed=-1, inverted    = 0x340BC6D9
```

* also known as CRC16-ARC
** also known as CRC16/CCITT, CRC16/V-41-LSB, CRC16-KERMIT
*** also known as CRC16-Modbus, if the two bytes are swapped i.e. 0x374b

For a detailed treatment of the many CRC algorithms known to exist, see
https://www.lammertbies.nl/comm/info/crc-calculation and
https://reveng.sourceforge.io/crc-catalogue/16.htm

As can be seen above, both CRC16_1 and CRC16_2 can be used for the Modbus CRC. To reverse the two bytes:

```
uint16_t i=crc;
crc>>=8;
crc+=(i<<8);
```

There are many other CRC algorithms. For example, none of the above CRC functions can produce a CRC identical to the CRC function in the KK Systems PPC or KD485-PROG products, the source code of which is available on request.

Random Number Generation

```
uint32_t RandVal (void);
```

Returns a random 32-bit number. This is a true random number, generated by a hardware generator in the 32F417. This function is also used by HTTPS/TLS and is thus mutex protected.

Execution time 5µs.

Timers

The KDE485 provides two kinds of timers:

Simple timers

These are in KDE_simple_timer.c and there are 20 of these, numbered 0-19. They are decremented at 1kHz, to zero. These are thread-safe i.e. one RTOS task can use timer 3 while another one can use timer 7.

```
int16_t loadtimer (uint8_t timernum, uint32_t timervalue);
```

Loads the timer; value in ms. Returns 0.

```
uint32_t readtimer (uint8_t timernum);
```

Reads the timer.

Example:

```
// A 1 second wait
loadtimer(3,1000);
while (readtimer(3)!=0) { }
```

RTOS timers

These are provided by FreeRTOS and are documented here
https://www.freertos.org/RTOS-software-timer.html
and there is more information in timers.h. The RTOS runs all timers in one task, with priority of configTIMER_TASK_PRIORITY (2).

The main advantages of these timers are

- a large number of timers can be defined
- there is a callback function which can run some code when a timer expires

Examples can be found in KDE_http_server.c and other KDE485 modules.

Wait for a very short delay

The above timer functions provide timers with a 1ms resolution. The following is for rare applications where one is communicating directly with hardware and where very short delays, of a few microseconds or less, are required:

```
void KDE_hang_around_us (uint32_t delay);
```

This waits for the specified time in microseconds. The accuracy depends on what other RTOS tasks are running, but this function will deliver a **minimum** wait. It is useful for ensuring that e.g. minimum device CS (chip select) times are met, and it will run even when interrupts are globally disabled. In most KDE485 applications this kind of coding will never be required.

Watchdog

The 32F417 processor contains two hardware watchdog timers: the window watchdog (WWDG) and the independent watchdog (IWDG). The KDE485 uses the IWDG.

The default configuration (set in KDE_main.c) is retriggered from a 1kHz timer interrupt, with a 1600ms timeout, so a basic crash protection level is provided even if the user program does not use the watchdog. For example, an invalid instruction trap stops all code execution and the watchdog will thus reboot the KDE485. But just an infinitely looping program will not, but in such a case other RTOS tasks will continue running, though probably slower.

```
void KDE_watchdog_init (uint32_t timeout_ms);
```

Initialises the watchdog. The timeout_ms value (max 4095) specifies the time in ms within which it must be "pulsed", either by KDE485 system (mode 0) or by the user program (mode 1).

To enable recovery of user programs which inadvertently set the watchdog incorrectly (see Watchdog Gotchas below) this function returns with no action if the button is pressed and the hex switch is set to F.

```
void KDE_watchdog_set_mode (uint8_t mode);
```

Change the mode of the watchdog:

0 for system trigger (pulsed from the KDE485 timer tick, every 1 ms)
1 for user trigger (must be pulsed from user code; max period is 1600ms)

```
void KDE_watchdog_pulse (void);
```

Pulse the watchdog. User code must call this function every 1600ms or less (period configured with KDE_watchdog_init) otherwise the KDE485 restarts.

Watchdog Gotchas

Careful use of the watchdog helps to produce applications which cannot produce a "hung" KDE485. The typical usage can be found in the KDE_app.c demo application.

There are subtle situations where the watchdog can trip unexpectedly. Unsurprisingly they involve cases where the code which calls KDE_watchdog_pulse() is not able to run, or runs too slowly for the configured timeout.

A common reason is that the task priority in the RTOS is too low and another task stops your code running for a while. This should not happen with carefully designed code but one thing which is outside your control is a PC Host writing files to the KDE485 filesystem via USB. The PC usually performs a rapid series of FLASH writes; each sector takes around 15ms to program into the filesystem (an SPI serial FLASH with a 15ms sector program time) and during this time the KDE485 runs very slowly - the RTOS 1kHz task switching runs 15 times slower than normal. This is a consequence of the USB MSC (removable drive) implementation in the KDE485 having been done for maximum PC operating system compatibility. Little can be done to address this using RTOS task priorities.

You can also end up with a permanently resetting KDE485! If you enable the watchdog but the pulsing code is never executed, you have a "bricked" unit. If you have physical access to the unit then you can use an SWD debugger to load new customer code, or holding down the button at startup and the hex switch being set to F will disable the watchdog until the next startup. But if the KDE485 is at a remote location, you are stuck.

The watchdog does a hard CPU reset and this can corrupt the filesystem if it happens when a file is being written. It may then need a re-format, chkdsk /f or similar.

For special applications, the KDE485 can be factory-configured to start the IWDG in a mode where it cannot be subsequently disabled by software. This mode is not recommended but it can somewhat reduce the likelihood of certain problems which manifest themselves quickly after startup, before the user RTOS task which pulses the watchdog has a chance to get going. This mode requires that the code which pulses the watchdog starts within 250ms of KDE485 startup, and it requires modifications to KDE_main.c to eliminate delays such as the power-up LED sequence.

Mutexes and other FreeRTOS Features

A mutex is a mutual exclusion lock. Only one thread can hold the lock. Mutexes are used to protect objects or resources from concurrent access.

Most hardware features need to be protected in a multi-threaded (pre-emptive RTOS) system and one example is the hardware random number generator. In some cases mutexes are used with software which is not thread-safe e.g. the heap functions and the KDE_* functions for the FatFS filesystem. Accordingly, mutexes are used extensively in KDE485 system software.

In the KDE485, mutexes are provided by FreeRTOS. Details can be found online e.g. https://www.freertos.org/Documentation/RTOS_book.html

In KDE485 user programs, the main use of mutexes is likely to be for synchronising access to some data, shared between two or more RTOS tasks.

These two functions are the most useful ones, and are usually used with an indefinite timeout:

```
osStatus_t osMutexAcquire (osMutexId_t mutex_id, uint32_t timeout);
osStatus_t osMutexRelease (osMutexId_t mutex_id);
```

Example:

```
// initialisation
#include "cmsis_os2.h"
#define osWaitForever 0xFFFFFFFF
osMutexId_t mutex1;
mutex1 = osMutexNew(NULL);

// within your code
osMutexAcquire(mutex1, osWaitForever);
...code accessing the shared resource...
osMutexRelease(mutex1);
```

The above code will wait indefinitely in the function osMutexAcquire until the shared resource is unlocked. During this time other RTOS tasks can run.

Obviously, mutexes need to be used with care e.g. if a protected function contains an early return() the mutex release on exit will not happen. A more obscure problem is referred to in Computer Science academia as "priority inversion": a low priority task locks a resource and does not release it (for a long time, or ever) and prevents a high priority task accessing that resource.

FreeRTOS has a large number of other features which are not used in the KDE485 e.g. queues for inter-process communication.

Miscellaneous API Functions

uint8_t buffer[12];

```
void KDE_get_CPU_ID (uint8_t * buffer);
```

The buffer is filled with a 12 byte unique CPU ID. This ID does not relate to the KDE485 serial number.

```
uint32_t KDE_get_serial (void);
```

Returns the KDE485 serial number.

```
float KDE_read_CPU_temp (void);
```

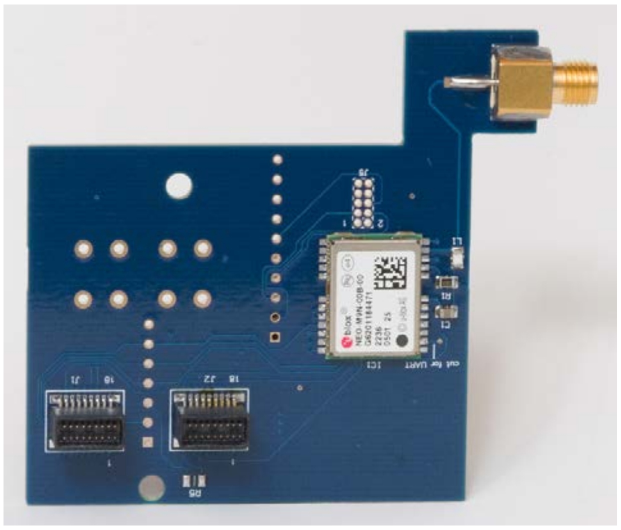
Returns CPU on-chip temperature. This is normally 20-40°C above ambient. It fluctuates according to the software running at the time. The maximum operating temperature is 105°C.

```
void KDE_reboot (void);
```

Reboots the KDE.

GPS

This section covers both a serially (RS232/422) connector GPS and the internal GPS factory option below



GPS Type Selection

The GPS interface, when enabled with gps_rtc in config.ini, runs as an RTOS task. The GPS type is selected according to gps_rtc:

0	no GPS
1,2	GPS connected to RS232 port 1 or 2
4	GPS connected to RS422 port 4
-1	GPS is the internal factory option (SPI-GPS)

GPS Functionality

In the KDE485, GPS serves two functions:

1) A source for **UTC date and time**. This data is extracted from the common G*RMC NMEA sentence and works with any GPS which outputs this sentence. Only the UTC date and time are extracted from this sentence. The KDE Real Time Clock (RTC) is periodically updated from this data.

2) A source for **GPS position**, and other data related to it. This data requires the GPS to be the U-BLOX NEO-M9N, or one of the similar U-BLOX models. Some U-BLOX proprietary sentences are used. A U-BLOX initialisation string is transmitted at startup to configure the GPS to deliver the required sentences at 5Hz.

The initialisation string is transmitted every 2 seconds until the required data is seen. The string does **not** save the new configuration to the GPS's flash memory.

The data is made available via the structure "gps".

The following U-BLOX sentences are used:

PUBX00 - used for position i.e. lat, long, alt, gs, track, hacc, vacc, hdop, vdop, tdop, #sats, fix type (NF, G2, G3, D3 etc)
PUBX03 - used for SBAS satellite detection
G*RMC - used for date/time (not proprietary to U-BLOX)

The G*RMC is alone sufficient for date/time. This enables a generic GPS, connected on a serial port, to support the NTP Server function.

See the notes on **NTP Server** for examples of the gps_rtc parameter for config.ini. Normally, if a GPS is in use, set ntp_server=0.0.0.0 (the address of the NTP server on the internet) to stop the NTP Client trying to get time from the internet. However, the NTP client is disabled from setting the RTC if GPS time is available.

GPS API (U-BLOX NEO-M9N)

If using a U-BLOX NEO-M9N GPS, position data becomes available:

```
bool get_gps ( struct gps * mygps, bool wait_mode, uint32_t timeout );

mygps               "gps" structure defined below
wait_mode           true:  waits for "valid gps", with the timeout, fills in the
                           structure, and returns true
                    false: returns the "valid gps" flag immediately
timeout             timeout in ms for the "true" mode above (2000 recommended)

struct gps
{
      uint32_t gps_date;              // date in ddmmyy
      uint32_t gps_utc;               // 24h UTC time in ms
      double gps_latitude;            // in degrees
      double gps_longitude;           // in degrees
      double gps_altitude;            // in metres
      uint8_t gps_n_s;                // 'N' or 'S'
      uint8_t gps_e_w;                // 'E' or 'W'
      uint8_t gps_fix[2];             // 'NF', 'D3' etc (two ASCII characters)
      float gps_hacc;                 // HACC (hor. accuracy)
      float gps_vacc;                 // VACC (vert. accuracy)
      float gps_gs;                   // hor. velocity (ground speed)
      float gps_vvel;                 // vert. velocity
      float gps_track;                // track
      float gps_hdop;                 // HDOP (hor. dilution of precision)
      float gps_vdop;                 // VDOP (vert. dilution of precision)
      float gps_tdop;                 // TDOP (time dilution of precision)
      uint16_t gps_num_sats;          // # of sats used in the nav solution
      uint16_t gps_sbas_num;          // # of SBAS sats being received (max 1)
      uint16_t gps_sbas_total_seen;   // # of SBAS sats seen since startup
};
```

For detailed info on the HACC etc values, see the NEO-M9N manual.

There are various complications around the GPS fix and SBAS values. These relate to US WAAS sats offering not just SBAS augmentation but also positioning, whereas the European EGNOS sats do just SBAS augmentation. The result is a lack of a close relationship between G3/D3 in gps_fix and the SBAS status.

GPS date/time becomes available with just 1 satellite, and tends to work well even indoors or with a poor antenna, whereas 3-4 are needed for position data.

Given that GPS data may not be available for various reasons, and very likely won't be available anyway for some time after GPS receiver startup, the wait_mode option is provided to enable two ways to structure your code.

With wait_mode=false, get_gps can just poll the GPS status:

```
if ( get_gps( &mygps, false, 0 ) == false )
	... do something else
```

or

```
if ( get_gps( &mygps, false, 0 ) == true )
{
	get_gps( &mygps, true, 2000);
	latitude = mygps.gps_latitude;
	longitude = mygps.gps_longitude;
	etc
}
```

With wait_mode=true, get_gps can wait, with a specified timeout, until valid GPS data appears:

```
if ( get_gps( &mygps, true, 2000 ) == true )
{
	latitude = mygps.gps_latitude;
	longitude = mygps.gps_longitude;
	etc
}
else
	... GPS timed out, so do something else
```

Or you can just wait for ever:

```
get_gps( &mygps, true, 1000000 );
```

Any waiting doesn't affect other RTOS tasks running.

The GPS task uses timers 20-29 internally.

The U-BLOX NEO-M9N was chosen because it is a modern high performance unit, is new and should be available for many years, and is very cheap. It is used in the internal GPS factory option. It is also available boxed from many vendors, with an RS232 interface.

LED Functions for GPS

If led_comms=0 in config.ini, the LEDs indicate the following:

LED 0	GPS data arriving
LED 1	2D fix
LED 2	3D fix
LED 3	SBAS satellite(s) being received
LED 4	Output data generated

RTC Real Time Clock

The RTC is a hardware implementation with its own power backup. The backup is a "supercap" capacitor which is sufficient for approximately 3 days. Unlike all types of rechargeable cells this is not life-limited.

Accuracy

The accuracy is the accuracy of a standard 32768Hz watch crystal - around 20ppm i.e. around 10 minutes per year. In some applications this is not adequate and there are 3 ways to obtain date/time for setting the RTC:

1) From user application code which obtains the date/time (e.g. via a serial port)
2) From GPS (see **GPS** section for automatic RTC setting)
3) From NTP client (internet time - configured in config.ini)

Two functions are provided for setting and reading the RTC: **setrtc** and **getrtc**. Both use the standard C tm structure:

```
struct tm
{
      int   tm_sec;     //    seconds after the minute  (0..59)
      int   tm_min;     //    minutes after the hour    (0..59)
      int   tm_hour;    //    hours since midnight      (0..23)
      int   tm_mday;    //    day of the month          (1..31)
      int   tm_mon;     //    months since January      (0..11)
      int   tm_year;    //    years since 1900          (100..199)
      int   tm_wday;    //    days since Sunday         (0..6)
      int   tm_yday;    //    days since Jan 1          (0..365)
      int   tm_isdst;   //    Daylight Saving Time flag (-1, unimplemented)
};
```

The value tm_wday is ignored by setrtc which calculates its own value from the date, and the calculated value is loaded into the RTC. This implementation is dictated by the fact that many date sources do not give the day of week.

The value tm_yday is ignored by setrtc because it is not maintained by the hardware. It is calculated from the date and filled-in when getrtc is called.

Setting the RTC

```
int setrtc (struct tm * mytime);
```

Returns:

0 no errors
1 RTC not present (not implemented)
2 hardware error (should never happen)
3+ for various illegal input values

Note that tm_year is in years since 1900 as per the C standard, but the RTC hardware holds just 2 digits for the year, and since 2000 was a leap year but 1900 was not, it is assumed that the RTC's automatic leap year adjustment works only for years after 2000.

The daylight savings flag is unimplemented because this is basically impossible on an embedded system. Any implementation needs internet access for the online updates required to handle the ~200 time zones. Accordingly, the RTC is usable only for UTC, or only some form of "local time".

The function zeroes the subseconds (g_SubSeconds) value.

Reading the RTC

```
int getrtc (struct tm * mytime)
```

Returns:

0 if no errors
1 if RTC not present (not implemented)
2 hardware error (should never happen)

The function also returns the subseconds value in a global variable g_SubSeconds. This is in microseconds, uint32_t, with a value from 0 to 999999. The resolution is 1/1024 of a second.

The value tm_wday is read from the RTC hardware and if it was incorrectly set at some earlier point then a wrong value will be returned. The setrtc function computes the value upon loading the RTC but the RTC hardware does not subsequently check it; it merely increments it at 00:00:00 each day.

The value tm_yday is not maintained by the RTC hardware. It is calculated from the date and filled-in when getrtc is called.

The daylight savings flag is unimplemented for same reason as setrtc above.

The RTC functions setrtc() and getrtc() are thread-safe.

The timestamps on files written to the flash file system by an internal application will be those obtained from getrtc().

RTC Battery Backup and Vbat

A 0.22F supercap powers the RTC for around 3 days. It takes a few minutes to fully charge. The KDE485 obviously does not normally initialise the date/time at startup otherwise the backup would be pointless. It does so if there is an indication that the supercap has discharged. This indication is implemented in two ways:

A unique value is written into a small memory area which is a part of the RTC hardware. If this is found to be corrupted, it is assumed the RTC time is corrupted too. This is obviously not 100% reliable because this value may or may not be lost at the same point at which the RTC oscillator stopped running.

Vbat (see below) is measured at KDE485 startup and, if found to be below 1.8V (the minimum Vbat spec for the RTC is 1.65V) the RTC is initialised. This is more reliable, unless the KDE485 had been subjected to some bizzare power cycling.

When the date/time is initialised, it is set to approximately the build date of the firmware.

Therefore, any application which needs date/time needs to have a means of setting the RTC at startup, using NTP, GPS, or some other external source.

Reading Vbat

The supercap voltage, Vbat, can be obtained, in volts, with

```
float KDE_read_Vbat (void)
```

The minimum specification for the RTC to run is 1.65V.

ARINC429

The ARINC429 subsystem is a factory installed **hardware** option. An API is provided to facilitate easy development of custom ARINC429 products.

The controller is a HI3593 which has two inputs and one output, but on the standard KDE product, one input and one output are accessible on the screw terminals. These replace serial port 4, using its four connections plus GND. The second input is accessible on an internal connector.

Startup code checks for the presence of the HI3593 (using its internal loopback feature) and, if present, sets a flag which is returned by

```
bool KDE_arinc_installed (void);
```

Both ARINC429 speeds are supported and are individually selectable on each channel, with:

```
uint8_t KDE_ARINC_HI3593_speed (uint8_t channel, uint8_t speed);
```

Channel:
0 = TX
1 = RX1
2 = RX2

Speed:
0 = low speed (12.5 kbps)
1 = high speed (100 kbps)

Returns:
0 if no errors
1 if HI3593 not present
2 if channel invalid
3 if speed invalid

Write data to HI3593:

```
bool KDE_ARINC_write (uint8_t opcode, uint8_t numBytes, uint8_t *buffer);
```

This function is HI3593 specific and issues an opcode (consult the HI3593 data sheet) to write the number of bytes from the buffer.

Returns true if successful.

Example:

```
uint8_t txdata[4];
// Get the data into bytes 0 1 2 3 where txdata[3] is the label
txdata[0] = ( (pkt_value>>24) & 0xff ) | ( ssm << 5 );
txdata[1] = (pkt_value>>16) & 0xff;
txdata[2] = ( (pkt_value>>8) & 0xff )  | 0x01;  // SDI = 01
txdata[3] = oct_label;
KDE_ARINC_write(0x0c, 4, txdata);
```

The HI3593 has a 32-packet transmit (TX) queue and a 32-packet receive (RX) queue. These must not overflow.

TX rate can be managed by protocol timing, TX code timing, or by checking TX status and waiting if full.

The first two methods are the best way; for example if converting from a 5Hz GPS NMEA data to ten ARINC429 packets, it is impossible to fill up the TX queue even at the slow ARINC429 speed because the output rate is just 50 32-bit packets per second which is around 1/7 of the 12.5kbps max rate. The last method - waiting on the TX full status - is generally the least preferred because it likely implies that some input data is being discarded.

Read data from HI3593:

```
bool **KDE_ARINC_read** (uint8_t opcode, uint8_t numBytes, uint8_t *buffer);
```

This function is HI3593 specific and issues an opcode (consult the HI3593 data sheet) to read the number of bytes to the buffer. The buffer must be large enough to receive the number of bytes specified. The data being read must be already present in the HI3593 RX queue; use KDE_ARINC_RX_status() to check.

Returns true if successful.

Example:

```
uint8_t rx1data[4];
KDE_ARINC_read(0xa0, 4, rx1data);   // get RX1 message
```

Get RX status:

```
bool KDE_ARINC_RX_status (uint8_t channel);
```

Returns true if there is at least 1 packet in the RX queue, and the channel (1 or 2) is valid.

This function is thread-safe, allowing two RTOS threads to service the two RX channels, provided that the two threads use different channels.

Get TX space status:

```
bool KDE_ARINC_TX_status (uint8_t channel);
```

Returns true if there is room for at least 1 packet in the TX queue, and the channel (1) is valid.

Get TX empty status:

```
uint8_t KDE_ARINC_get_TXE (void);
```

Returns true if TX queue is empty, and the channel (1) is valid.

Transmit ARINC429 packet (specific labels supported):

```
uint8_t KDE_ARINC_TX_PKT (uint8_t channel, uint16_t label, double value, uint8_t
ssm);
```

This function is provided for convenience, and is intended for most common ARINC429 transmit operations, where a value representing some analog parameter is transmitted. The way in which the value is encoded depends totally on the label. Labels currently supported are 103, 112, 136, 247, 320. More are being added.

Channel must be 1 because HI3593 has only the one TX channel.

Value is the value to transmit (usually it ends up being sent as a signed integer).

SSM is normally 3 ("good data") if the value being sent out is a binary value (most of them are)

Returns:
0 if no errors
1 if HI3593 not present
2 if channel invalid
3 if label invalid per ARINC429 (0 or >377)
4 if label not supported by this function
5 if SSM invalid

Transmit ARINC429 packet (GPS group):

```
uint8_t KDE_ARINC_TX_GPS (uint8_t channel, double latitude, double longitude,
uint8_t ssm);
```

This function is provided for convenience, and is intended to transmit the entire ARINC429 "position" group comprising of the six labels

110 GPS latitude
120 GPS latitude fine
111 GPS longitude
121 GPS longitude fine
310 GPS latitude
311 GPS longitude

The six labels are grouped into one function and are transmitted all together because 110+111 each carry a part of a 31 bit (+sign) integer representing latitude, in units of ~8.38E-8 (of 180 degrees), and the split must be done without the value changing before/after the split. 120+121 do the same thing for longitude. 310,311 are the same structure as 110,111.

Channel must be 1 because HI3593 has only the one TX channel.

SSM is normally 3 ("good data").

Returns:
0 if no errors
1 if HI3593 not present
2 if channel invalid
3 not used
4 not used
5 if SSM invalid

Before calling this function, make sure there is room in the HI3593 TX queue for the six packets. The queue can hold 32 packets. The room can be ensured by timing design, or by checking KDE_ARINC_get_TXE().

Example:

```
// Transmit GPS "position" group
KDE_ARINC_TX_GPS(1, mygps.gps_latitude, mygps.gps_longitude, 3);
// Transmit track, ground speed, V and H accuracy
KDE_ARINC_TX_PKT(1, 103, mygps.gps_track, 3);
KDE_ARINC_TX_PKT(1, 112, mygps.gps_gs, 3);
KDE_ARINC_TX_PKT(1, 136, mygps.gps_vacc*3.28084, 3);       // vfom in ft
KDE_ARINC_TX_PKT(1, 247, mygps.gps_hacc*0.000539957, 3);   // hfom in nm
```

Note that the latitude and longitude are supplied in the standard GPS NMEA format i.e. 0 to 359.999 degrees, while the ARINC429 transmitted value is from -179.999 to +179.999 degrees. The foregoing precision (0.999) is a simplification because the value is represented on ARINC429 as an integer and the code was carefully written to limit the value sent out to exactly the range of that integer.

Receive ARINC429 packet (specific labels supported):

```
uint8_t KDE_ARINC_RX_PKT (uint8_t channel, uint16_t *label, double *f_value,
int32_t *int_value);
```

This function is provided for convenience, and is intended for most common ARINC429 receive operations, where a value representing some analog parameter is received. The way in which the value is encoded depends totally on the label. Labels currently supported are 203 and 320. More are being added.

This function is thread-safe, allowing two RTOS threads to service the two RX channels, provided that the two threads use different channels.

Function waits (with a simple timeout) until data is received so you should call KDE_ARINC_RX_status() first.

Channel is 1 or 2.

Returns:

0 if no errors
1 if HI3593 not present
2 if channel invalid
3 if label invalid per ARINC429 (0 or >377) and returns the received label value
4 if label not supported, and returns the received label value
5 if timeout
6 if parity error
7 if SSM invalid (should be 3 for binary values following)

Label: ARINC429 label
f_value: parameter value as a double float
int_value: actual integer parameter (cleaned up into int32)

SPI RAM (8MB)

This is a factory installed **hardware** option. It can reside on the main KDE485 board or on the SPI-GPS board.

The storage is block-oriented and is not normal linear RAM in the CPU address space.

In the following functions, address is a linear address 0-8MB (0-0x7FFFFF) within the memory device, and buf is the user's data buffer and must be a static address (not on the stack) because DMA is used for maximum performance which reaches 2MB/sec. The return value is **true** if the RAM device is installed.

There are limitations on the duration of the chip select signal (effectively, on the transfer size in bytes) which relate to the internal refresh of the pseudo-static device being used, and these are temperature dependent. Within the KDE485's ambient temperature specification, a maximum transfer size of 512 bytes is recommended.

```
bool KDE_SPI_RAM_write (uint32_t address, uint8_t * buf, uint32_t length);
bool KDE_SPI_RAM_read (uint32_t address, uint8_t * buf, uint32_t length);
```

32F417 / 32F437 CPU option

The 32F417 has 1MB of FLASH. The lowest 32k is a "boot block" which user software never writes. This boot block contains startup code and factory software recovery code. The only time this boot block is rewritten is when the factory software is loaded.

Then around 400k implements the functionality and API described in this manual. The remainder of the 1MB is available for user code. This can be expanded; another 200k can be freed-up by e.g. excluding TLS from the build (see **Unreachable Code Removal**).

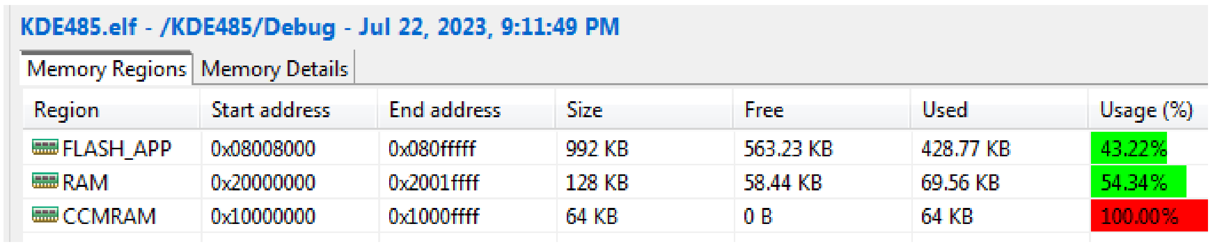
The 32F417 256k total RAM is in two parts: 128k conventional RAM and 64k CCM. The 128k is used for

- stack for ISRs (8k)
- static storage
- the heap (used mostly for TLS; 48k)
- main.c stack usage prior to starting RTOS

and there is around 50-60k remaining for user applications, after the full factory software is built.

The 64k CCM is used for the FreeRTOS task stacks. The linker script is set up to use up 100% of this memory, so it always shows in RED below.

The following Build Analyser display in Cube IDE shows a typical memory usage picture, showing 563k and 58k available for user FLASH space and user RAM space, respectively. The 428k used represents the full factory build containing all the features described in this manual.



KDE485.elf - /KDE485/Debug - Jul 22, 2023, 9:11:49 PM

Memory Regions | Memory Details

Region	Start address	End address	Size	Free	Used	Usage (%)
FLASH_APP	0x08008000	0x080fffff	992 KB	563.23 KB	428.77 KB	43.22%
RAM	0x20000000	0x2001ffff	128 KB	58.44 KB	69.56 KB	54.34%
CCMRAM	0x10000000	0x1000ffff	64 KB	0 B	64 KB	100.00%

For applications requiring more RAM, the KDE485 can be built with a **32F437** which has another 64k of RAM. This extra 64k is usable for both statics and the heap. The 32F437 is auto detected at runtime; Cube IDE configuration remains at 32F417 and the Build Analyser display does not change. Contact Factory regarding availability.

The actual CPU type can be found in a global variable

uint32_t g_dev_id;

which is 417 or 437 (decimal). It is also reported in the Status page in the HTTP server.

If a 32F437 is being used and the full extra 64k RAM is to be used for statics, and a representative Build Analyser display is desired, change the RAM ... LENGTH=128K to 192K in the LinkerScript.LD file. That also prevents linker errors if the extra RAM is used for statics.

ANALOG SUBSYSTEM

The KDE485 has three analog sections:

1) **Standard**: 0-20mA current sink / relay drive, 2 x 12-bit ADC, 2 x 12-bit DAC

2) **Optional on (on board)**: PT100/PT1000, thermocouple, 4-20mA input, voltage input

The circuitry is built with precision components for long term stability and undergoes factory calibration to achieve high accuracy.

3) **Optional (on add-on board)**: 22-bit ADC

This uses an MCP3550 ADC and resides on the optional GPS board. The SMA connector normally used for the GPS antenna is now used for the analog input.

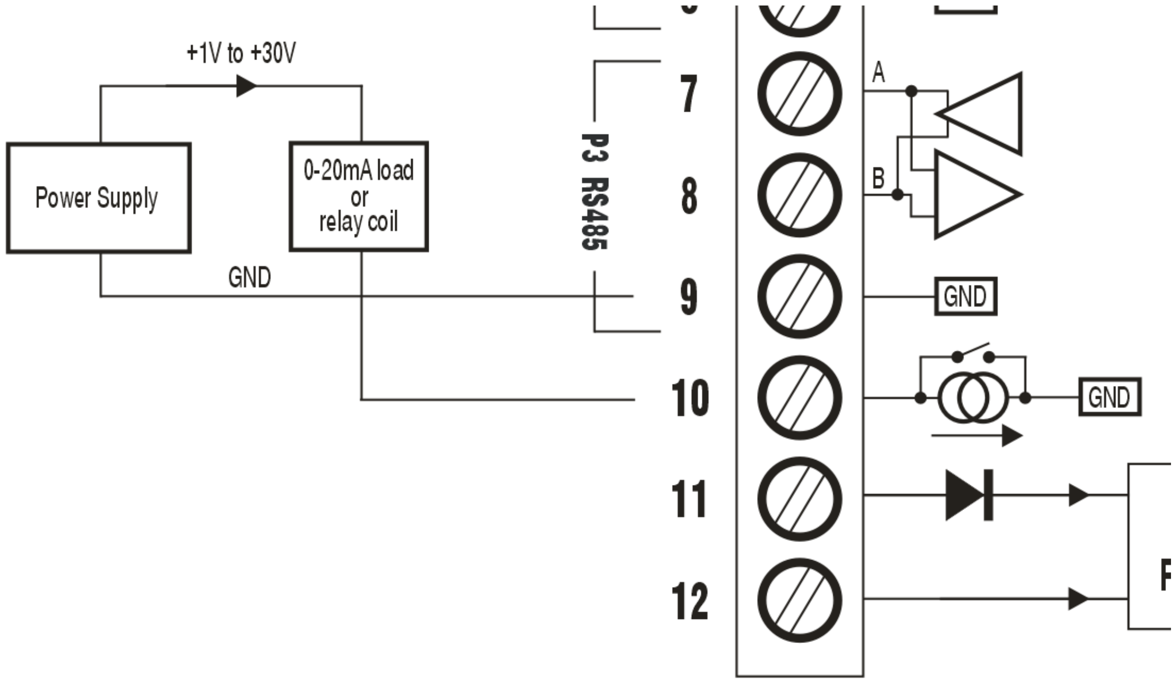
Standard Analog Subsystem (non isolated)

This has two applications: emulate a 4-20mA sensor, and drive a relay.

4-20mA Current Sink for 4-20mA Sensor Emulation / Relay Drive

The same pin (pin 10, connector 1) is used for both 0-20mA and the relay drive.

The output (pin 10) is an open-drain MOSFET



and the current is sunk into the KDE485 ground (GND above). This subsystem is isolated from the KDE485 power supply (pins 11,12) but is not isolated from the four serial ports.

The power supply above can be the same one as is used to power the KDE485 (pins 11,12) but then you lose the isolation between the KDE485's power input and the rest of the KDE485, notably its serial ports.

For 0-20mA applications (in this context this means the KDE485 is emulating a 4-20mA sensor) the power supply will normally be 24V, although some 4-20mA-sensor instruments (shown as "load" above) can work with lower voltages. The KDE485 can sink 22mA while dropping < 1V.

For relay drive applications the power supply will be according to the relay coil requirements. The absolute maximum is 30V; pin 10 is clamped to +36V with an internal zener diode. The maximum relay coil current is 100mA.

The following functions provide the interface to user applications:

```
void KDE_DAC_set_relay (bool state);
```

state: true to turn on relay
false to turn off relay (enables 0-20mA mode with KDE_0_20mA_out)

```
void KDE_0_20mA_out (float value);
```

value: 0.0-20.0 in mA

Note that if you call KDE_DAC_set_relay(false) and then call KDE_0_20mA_out(20) and there is a relay connected, the relay may well get turned on!

The current sink is factory calibrated to a high accuracy (see **Specification**) at 1%, 20% and 100% of range (0.2mA, 4mA and 20mA). Accuracy below 1% (0.2mA) is not guaranteed. There is an over-range capability of around 102% (20.4mA).

2 x 12-bit ADC, 2 x 12-bit DAC - on internal connectors

These are accessible on internal connectors only, for custom designed expansion boards. Contact Factory for more information. These peripherals are very fast - microseconds. All share a common 2.5V 1% reference (+Vref) and their input and output swings 0 to +Vref.
Both ADCs are used to monitor, via resistor dividers which can be overriden, the +5V and +3.3V supply rails and these are displayed in the Status page of the HTTP server. ADC1, if unconnected, measures 1/11 of the 5V rail, and ADC2, if unconnected, measures 1/2 of the 3.3V rail.

ADC1 is also used for the power fail data save feature. This measures the +5V rail and triggers the data save when this falls below +4.2V. This feature is disabled by default.

```
float KDE_CPU_Vref (void);
```

Returns the 2.5V reference voltage (+Vref, in volts) used for the two 12-bit ADCs and two 12-bit DACs. This enables user applications to use the 12-bit peripherals more accurately. The 12-bit ADCs and DACs straightforwardly scale 0-4095 to 0 to +Vref so if you know +Vref you can extract the maximum accuracy out of them. The production tolerance on Vref is up to 2% and this function returns the actual value (measured to <0.01%). See the STM32F417 data sheet for more information.

Read ADC1 or ADC2:

```
void KDE_ADC_ST_read_cal (uint8_t channel, float * value);
```

channel:	1 or 2
value:	0 to +Vref

Returns the input voltage 0 to +Vref. This value is already calibrated to +Vref. The result is undefined with an input voltage > +Vref.

```
void KDE_DAC_set_value (uint8_t channel, uint16_t value);
```

channel:	1 or 2
value:	0 to 4095

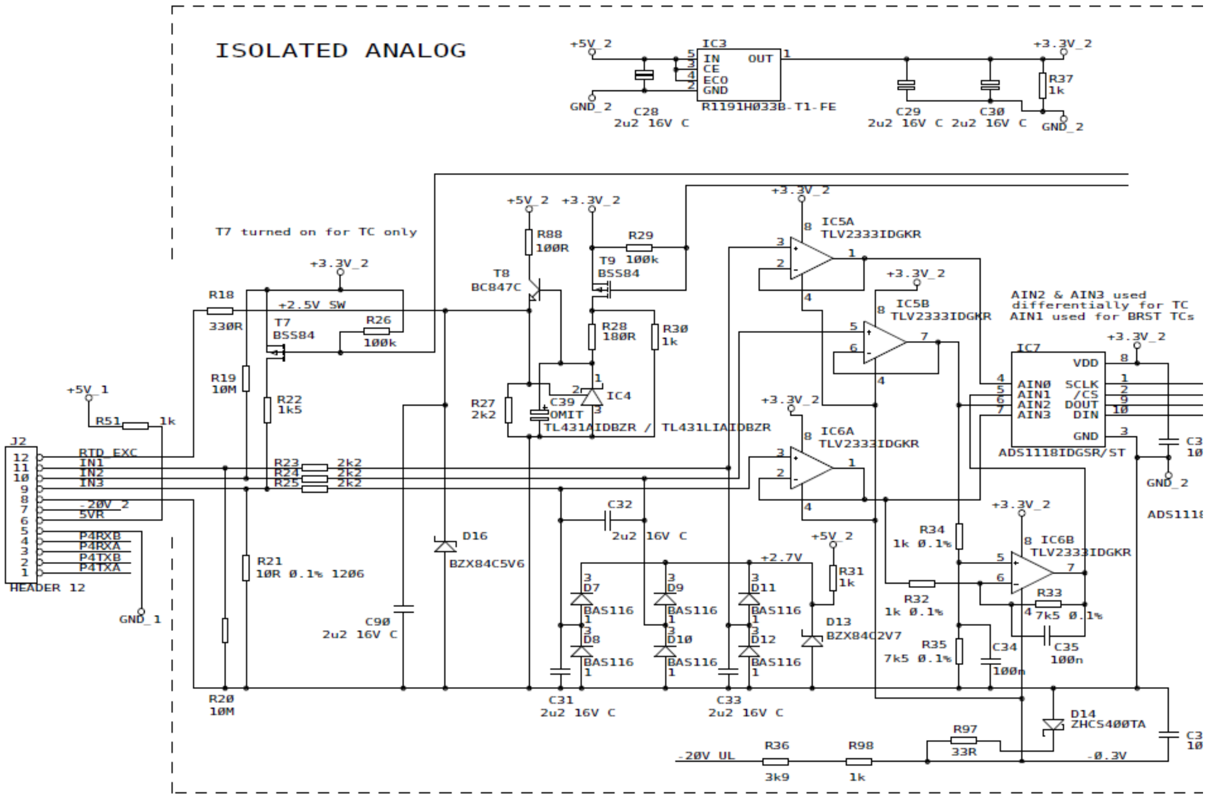
Sets the DAC channel to 0 to +Vref for inputs 0 to 4095. If value > 4095, it is clamped to 4095. This function is uncalibrated and KDE_CPU_Vref() can be used to improve the output accuracy if a specific voltage is required.

Optional Analog Subsystem (isolated, on-board)

Overview

The isolated analogue subsystem supports the following inputs

- PT100/PT1000 (-200°C to +850°C)
- all 8 thermocouple types B E J K N R S T (temperature ranges vary)
- 4-20mA sensors (powered from external +24V, or from internal -20V)
- AD590 sensors (µA proportional to Kelvin)
- general voltage measurement (256mV 512mV 1024mV 2048mV 4096mV) - 2 inputs



A high performance 16 bit bipolar delta-sigma ADC is used - the ADS1118. Sample time is user selectable from 1/860s to 1/8s. This also contains the CJC temperature sensor.

Input is filtered with an RC filter (2k2+2.2µF) with a time constant of 4ms. The settling time to 16 bits for a step input is 11 time constants.

The input ranges are unipolar only (positive voltages). Inputs are clamped to approximately +3V and -0.6V. For custom applications there is a facility for reading voltages down to -0.2V using IC6B and AIN1.

There are two ways to read the sensor value (or the input voltage):

- an RTOS task "SENSORS"
- direct API reading

Reading Sensors with an RTOS Task

An RTOS task "SENSORS" is enabled with **sensor_read=1** in config.ini. It is controlled by the following config.ini values:

```
sensor_read=1                  ; 1 enables RTOS task, 0 disables it
sensor_type=1                  ; 1=PT100 4-wire, 2=PT100 3-wire, etc; see table below
sensor_adc_sample=1            ; 1=1/8 sec ADC sample time; see table below
sensor_wait=1000               ; inter-reading gap in ms, 10 to 2^31
sensor_2w=0.0                  ; RTD 2 wire mode total wire resistance in ohms
sensor_out=0                   ; Output values to serial port 0-4; other values disable output
```

The data can be obtained

- via the HTTP Server **Analog** page
- via a serial port (using the sensor_out value in config.ini)
- via global variables below

The task loads the result in the following global variables, appropriately to each sensor type:

```
  bool g_sensor_valid (set to true when a reading has been performed, false on an error)
  float g_sensor_temp (sensor temperature in °C , for RTD or thermocouples, or AD590)
  float g_v_in1_ma (voltage input in volts (IN1), or 4-20mA current in mA
  float g_v_in2 (voltage input in volts (IN2)
```

Note that g_sensor_valid can go false for up to several hundred ms, if long ADC sampling times are used.

sensor_type

0	invalid
1	PT100 4-wire
2	PT100 3-wire
3	PT100 2-wire (sensor_2w = cable resistance in ohms)
4	PT1000 4-wire
5	PT1000 3-wire
6	PT1000 2-wire (sensor_2w = cable resistance in ohms)
7	thermocouple type B
8	thermocouple type E
9	thermocouple type J
10	thermocouple type K
11	thermocouple type N
12	thermocouple type R
13	thermocouple type S
14	thermocouple type T
15	4-20mA sensor, powered from internal -20V supply
16	4-20mA sensor, powered from external +24V supply
17	AD590 type sensor (μA = °K - needs an external 1k 0.1% resistor)
18	reserved (22 bit ADC option)
19	reserved
20	voltage input

Types 3 and 6 above are not accurate, due to the strong temperature coefficient of resistance of copper cable.

sensor_adc_sample

The ADS1118 ADC supports a range of sample periods. Due to KDE485 input filter settling time and other factors, the only ones which are worth using are 1/8 (125ms) and 1/128 (8ms); the others may be useful in specialised applications

0	invalid	
1	**1/8 sec**	total sample time 141ms
2	1/16 sec	total sample time 72ms
3	1/32 sec	total sample time 37ms
4	1/64 sec	total sample time 20ms
5	**1/128 sec**	total sample time 11ms
6	1/250 sec	total sample time 7ms
7	1/475 sec	total sample time 6ms
8	1/860 sec	total sample time 4ms

If sensor_wait is set to >= 10, add 100ms to the above sample times for input filter setting time.

sensor_wait

This sets the time period (in ms) between readings i.e. how often the RTOS task runs. Range is 10ms to 2^31ms. A value below 10 is read as 10. The config.ini file is checked each time for any changes and this takes about 30ms. The reading rate for is thus set by the combination of

- sensor_wait
- 100ms input filter settling time (if sensor_wait>=10)
- ADC sample time
- 30ms

Example: if sensor_wait=100, sensor_adc_sample=1 (sample time 141ms) , the reading will be updated at 1/0.368 i.e. approx 3Hz.

The fastest reading rate is with sensor_wait=0, sensor_adc_sample=8, and thermocouple sensors; this produces a reading rate of 20Hz. If a higher rate is required, please contact factory.

This parameter is especially relevant for for **PT100/PT1000** sensors. These sensors are powered only while they are read, unless sensor_wait < 10 and then they are powered continuously. These sensors exhibit self heating due to the current being passed through them, and reading them at a low duty cycle can be used to reduce this to a negligible amount, at the expense of a slower reading rate. The self heating error is strongly related to the size and construction of the sensor. See the Sensor Self Heating section. If the sensor_wait parameter is set to less than 10, the sensor power is applied continuously.

sensor_2w

This specifies the **total** cable resistance for RTD 2-wire mode, in ohms. This mode is not recommended because it yields poor accuracy, due to the temperature coefficient of copper wire, plus the unknown resistance. Note that the linearisation calculation will fail if sensor_2w is greater than the lowest possible sensor resistance (18Ω for PT100, 180 Ω for PT1000).

sensor_out

This parameter, if 0-4, causes the KDE485 to output the sensor data to the specified serial port. The value is in floating point, terminated by CRLF. The output is as follows

Temperature sensors:	Timestamp, temperature in °C
4-20mA sensors:	Timestamp, current in mA
AD590 sensors:	Timestamp, temperature in °C
Voltage:	Timestamp, IN1 and IN2 inputs in volts, separated by 1 space

The information is similar to that displayed in the HTTP Server Analog page but the timestamp is a more universal format . The data can be received by any device which can accept serial data, including a PC running a terminal application on a VCP.

Example:

```
20230817_213130  26.99C
20230817_213131  27.03C
20230817_213132  26.99C
```

Reading Sensors with the API

The following functions are provided for a fully customised solution, from your own RTOS task. In this case set **sensor_read=0** in config.ini to block the periodic sensor reading.

These API functions are **not** thread-safe.

Reading RTD (PT100/PT1000)

```
int KDE_read_rtd (int sensor_type, int sample, bool rtd_pwr,
                    float sensor_2w, float * temp);
```

Reads PT100/PT1000 RTD into g_sensor_temp, in °C

sensor_type:

1	PT100 4-wire
2	PT100 3-wire
3	PT100 2-wire (sensor_2w = cable resistance in ohms)
4	PT1000 4-wire
5	PT1000 3-wire
6	PT1000 2-wire (sensor_2w = cable resistance in ohms)

Types 3 and 6 above are not accurate, due to the strong temperature coefficient of resistance of copper cable.

sample:

1	1/8 sec
2	1/16 sec
3	1/32 sec
4	1/64 sec
5	1/128 sec
6	1/250 sec
7	1/475 sec
8	1/860 sec

rtd_pwr:

true	turns on RTD power, waits for settling time (only initially), performs conversion and leaves power on
false	turns on RTD power, waits for settling time, performs conversion and turns power off

sensor_2w:
total resistance in ohms of 2-wire RTD cable

Returns:

0	success
1	too cold (or short circuit)
2	too hot (or open circuit)
3	invalid sensor type / invalid sample value

Execution time of this function is x ADC sample times where x=3 for PT100 temperature below 290°C , or 4-5 for higher PT100 temperatures, or PT1000 sensors. On 1st call 100ms is added.

Reading Thermocouples

```
int KDE_read_tc (int sensor_type, int sample, float * temp);
```

Reads thermocouple into temp, in °C

sensor_type:

7	thermocouple type B
8	thermocouple type E
9	thermocouple type J
10	thermocouple type K
11	thermocouple type N
12	thermocouple type R
13	thermocouple type S
14	thermocouple type T

sample:

0	invalid
1	1/8 sec
2	1/16 sec
3	1/32 sec
4	1/64 sec
5	1/128 sec
6	1/250 sec
7	1/475 sec
8	1/860 sec

Execution time of this function is 2 ADC sample times, except on 1st call when 100ms is added.

Returns:

0	success
1	too hot, or open circuit
3	invalid sensor type / invalid sample value

Reading 4-20mA Sensors

As shown in the 4-20mA wiring section there are two ways to connect up a 4-20mA sensor: with a negative supply, or with a positive supply.

```
int KDE_read_4_20MA (int sensor_type, int sample, float * current);
```

Reads current in mA. Normally 4mA and 20mA correspond to process variable value of 0% and 100%, respectively.

sensor_type:

15	4-20mA sensor, powered from internal -20V supply
16	4-20mA sensor, powered from external +24V supply

Reading AD590 Sensors

```
int KDE_read_ad590 (int sensor_type, int sample, float * temp);
```

Reads sensor temperature in °C.

Sensor_type is always 17.

Returns:

0	success
1	too hot (above 512K)
3	invalid sensor type / invalid sample value

Reading IN1/IN2 Voltage

```
int KDE_read_volts (int sensor_type, int sample, float * in1, float * in2);
```

This is an autoranging DVM which reads both IN1 and IN2 inputs, as volts.

Sensor_type is always 20.

There is only one A-D converter so IN1 and IN2 are not read exactly concurrently.

Note that IN1 has a 10M pulldown to GND and IN2 has a 10M pullup to +3.3V. This affects readings with unconnected inputs, or inputs driven from a high impedance source.

Execution time of this function is 2 ADC sample times if both voltages are < 256mV, increasing by 1 ADC sample time for each voltage which is above 256mV, above 512mV, above 1024mV, above 2048mV, above 4096mV.

Input protection clamps the inputs to around +2.8V.

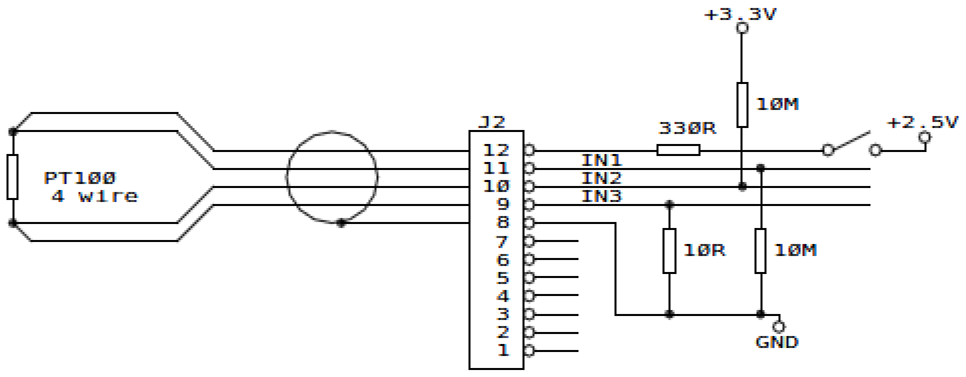
Sensor Wiring Diagrams and Limitations

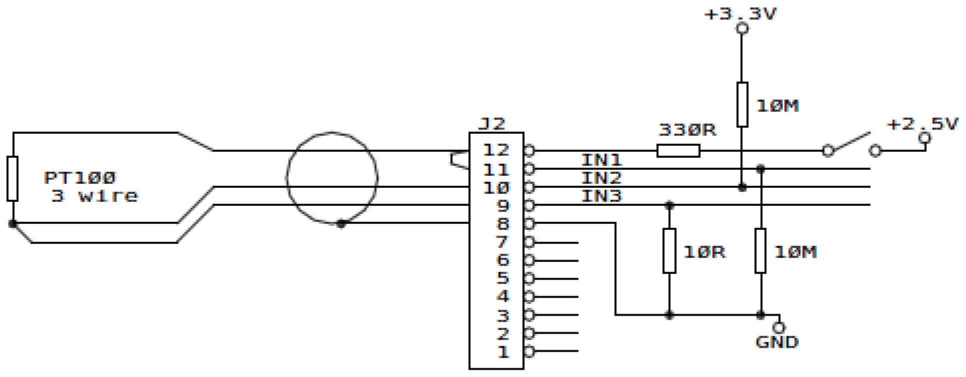
Note that on the KDE485 terminal blocks, pin 1 is on the left when the cable holes are facing you.

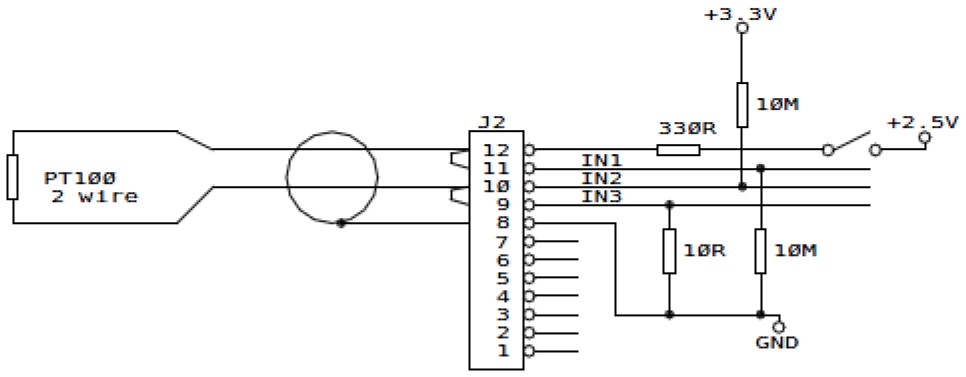
The 10R resistor shown in the diagrams below is factory calibrated to 0.01% equivalent tolerance.

RTD (PT100/PT1000) - sensor_type=1-6

This is a platinum sensor which offers the highest accuracy. The resistance range is based on the standard -200°C to +850°C temperature range which translates to 18 to 390 ohms, or 180 to 3900 ohms, for PT100 and PT1000 respectively. Most practical sensors are temperature-limited by the cable material.







The highest accuracy is achieved with the 4 wire mode. The 2 wire mode should be avoided because the temperature coefficient of copper cable will dominate the errors on all but very short cables.

RTD Sensor Self-Heating

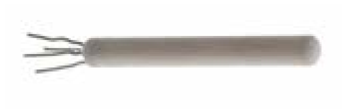
Experiments on PT100 sensor self heating have shown the self heating error can be reduced to negligible levels in most applications by carefully selecting the wait time between readings and the sample time. The following data was collected experimentally, for unmounted sensors in free air and for the specified sensor_wait value:

Conv time:	**1/125s**	**1/8s**

Sensor style 1: Smallest available sensor (2x3x0.5mm, unpackaged)

10s	<<0.1°C	0.1°C
3s	<0.1°C	0.4°C
1s	0.1°C	1°C
500ms	0.4°C	1.3°C
100ms	1°C	N/A

Sensor style 2: Cylindrical sensor (3mm diameter x 15mm, ceramic)



10s	<<0.1°C	<0.1°C
3s	<0.1°C	0.1°C
1s	<0.1°C	0.2°C
500ms	0.2°C	0.4°C
100ms	0.5°C	N/A

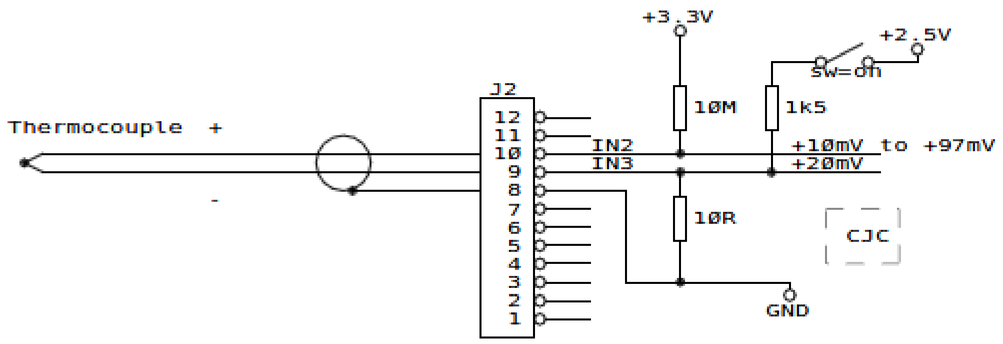
Sensor style 3: Cylindrical sensor (5mm diameter x 50mm, stainless steel)

Any	<0.1°C	<0.2°C

Relative to the above, note that under IEC 60751 Class A sensor accuracy is 0.15°C (0-100°C) and Class B is 0.3°C.

Thermocouple - sensor_type=7-14

This is the most popular temperature sensor. Except for the exotic types (type B,R,S platinum-rhodium) it is cheap and can be used for high temperatures. Due to its construction (a welded junction of two metals) and the need for cold junction compensation (CJC - difficult to do accurately especially with a removable terminal block) it is not particularly accurate.



The KDE485 supports all thermocouple types in known usage (output voltage ranges shown):

B	0 to +14mV
E	-10 to +77mV
J	-9 to +70mV
K	-7 to +55mV
N	-5 to +48mV
R	-1 to +22mV
S	-1 to +19mV
T	-7 to +21mV

By far the most common types are J and K.

Type B is non-monotonic below +50°C which means the output voltage from it cannot be unambiguously converted to temperature, in that range. Its output is very low so it is generally specified for +250°C to +1700°C and the KDE485 linearisation code regards anything below +250°C as a faulty thermocouple. In the HTTP server Analog display it is shown as "TC OPEN / below +250C". It is the highest temperature thermocouple and is extremely expensive: 3-4 figures for most practical packages.

In thermocouple mode, the KDE485 internally biases IN3 (pin 9) to +20mV, to ensure that the IN2 input (pin 10) remains above the analog ground (pin 8) even for a very cold thermocouple.

If a "grounded thermocouple" (where its negative wire, IN3, is connected to the sensor case and ends up accidentally connected to the analog ground) is used, this shorts out the 20mV bias. However, this should still work, with the limitation that temperatures below the cold junction may lose accuracy.

With most thermocouples, the white wire is the ground (pin 9 of the terminal block) and the non-white wire goes to pin 10.

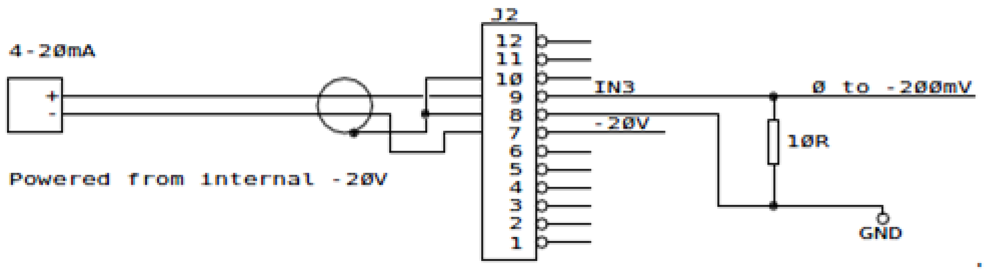
The CJC temperature is measured with a sensor close to the terminal block. This will not measure the exact temperature of the removable half of the terminal block. The KDE485 contains an internal adjustment of 2°C but the parameter **cjc_corr**=x.x can be used in config.ini to override this.

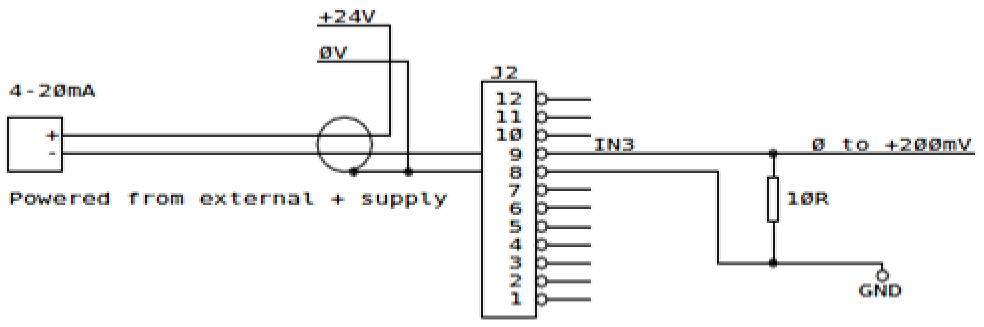
4-20mA - sensor_type=15,16

This is a popular sensor interface which is usually implemented by connecting a common sensor (e.g. a thermocouple) via a 4-20mA "conditioner" or "transmitter"

which fits inside a standard sensor housing. The main advantage of 4-20mA is that only 2 wires are needed and long cables can be used. The accuracy is OK for thermocouples but a PT100 loses a lot of its potential precision by going through a transmitter.

The KDE485 4-20mA mode supports sensors which sink current (sensor type 15 - powered from the internal -20V supply) or source current (sensor type 16 - powered from an external positive supply; typically +24V). The current produces a voltage drop across a 10 ohm reference resistor. This voltage is positive or negative according to which supply arrangement is used.





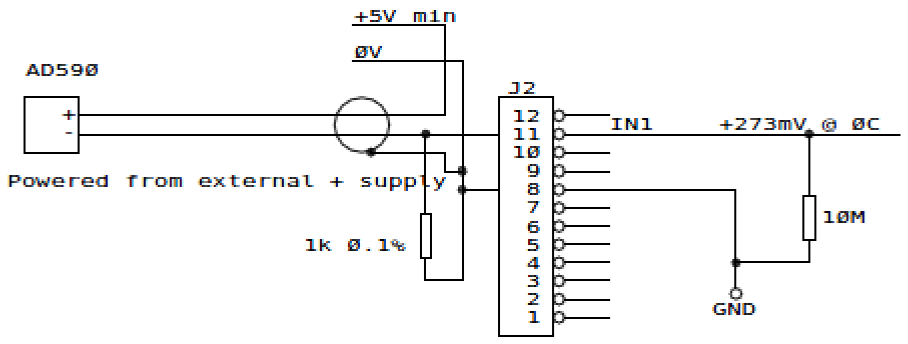
The KDE485 circuit is capable of under-range down to 0.2mA and over-range up to 22mA. Zero or negative current cannot be measured.

The -20V supply requires that the KDE485 is powered from **24V** (not 12V) and is internally limited to 30mA.

AD590 - sensor_type=17

This is a silicon sensor which draws a current in microamps equal to the absolute temperature (μA = °K). The AD590 designation was originally used by Analog Devices in the 1970s and these sensors exist under many names. They are widely used and the benefits are similar to the 4-20mA system, with the advantage of a much lower current drawn. However, only expensive versions have a reasonable accuracy, and all suffer from susceptibility to damage from electrical spikes (shielded wiring must be used) and long term drift at elevated temperatures e.g. above +70°C even on devices specified much higher.

An external 1k resistor (±0.1% usually suffices) is required. The measurement range used by the KDE485 is 0 to 512mV which is 0K to 512K which covers all such sensors. They generally need a supply of at least +4V which has to be provided externally.



Thermistor

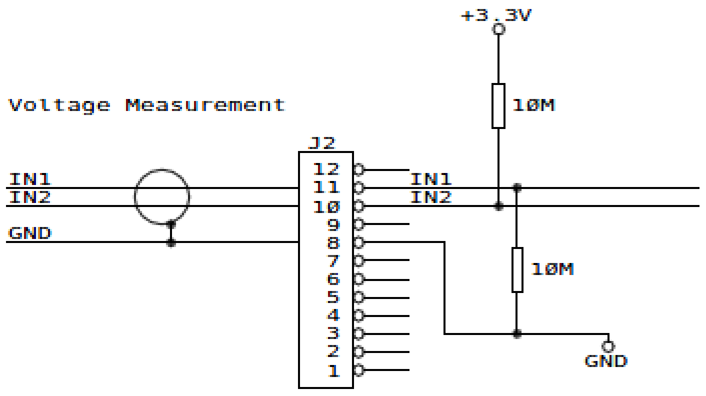
This sensor type does not have a dedicated provision in the KDE485 but would be connected in the same way as a PT100.

All thermistors are nonlinear but it is trivial to linearise them with a polynomial (the coefficients are available from the thermistor manufacturer) in the KDE485 C language.

Thermistors are available in many types and many packages, ranging from epoxy dipped (some of which are 0.2°C accurate but are limited to around +100°C) to glass encapsulated (less accurate but far more thermally robust.

Voltage Measurement - sensor_type=20

This is a two-input general purpose autoranging voltmeter



The circuit is intended for positive voltages. Inputs are clamped at approximately -300mV and +3.2V. Accuracy is maintained between 0 and +2.8V.

Note that IN1 has a 10M pulldown to GND and IN2 has a 10M pullup to +3.3V. This affects readings with unconnected inputs, or inputs driven from a high impedance source.

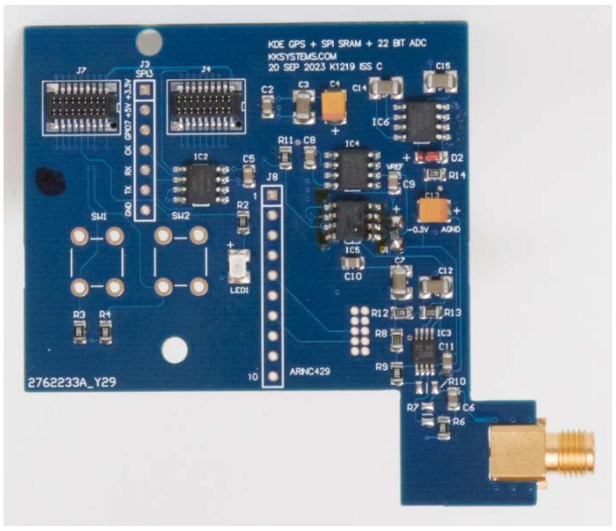
Isolation and Grounding

The entire optional isolated analog subsystem is isolated from the rest of the KDE485, with a design isolation of 2500V RMS and a test voltage of 1500V AC RMS which is 100% tested in production.

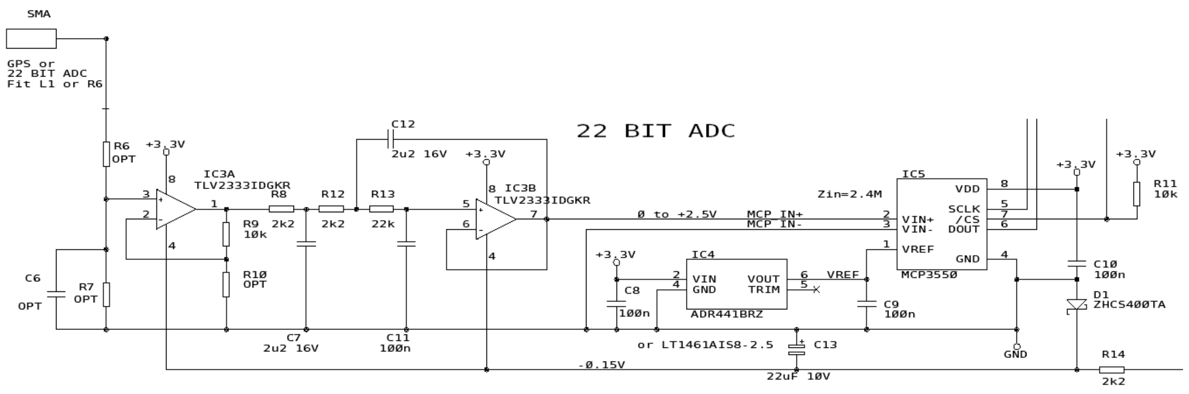
As with all isolated interfaces, to prevent static buildup, the analog ground should be connected to a real ground somewhere. Cable shields can be used for this purpose.

Optional 22 Bit Analog Subsystem (non-isolated, add-on board)

This uses the same PCB as the optional GPS. This option requires a mounting kit and is normally factory installed.



The SMA connector is used for the analog input.



The ADC is the MCP3550 and there is a buffer IC3A with a build-selectable gain (R6/R7 and R9/R10) and filtering (3rd order Sallen-Key filter around IC3B). The filter design can be conveniently done with an online filter calculator such as http://sim.okawa-denshi.jp/en/Sallenkey3Lowkeisan.htm.

The TLV2333 is a high spec chopper-stabilised op-amp. The AD441 reference is +2.5V with a 0.04% initial accuracy. Accordingly, on the standard build of this board, no factory calibration is performed. R7 and R10 are not fitted, creating an input range of 0 to +2.5V.

The API is just one function:

```
uint32_t MCP3550_read (int32_t* value);
```

Reads MCP3550 ADC. Execution time: 90ms.

Returns a 32 bit signed integer 'value' in the range of +/- 2097152 (2^21).
Returns a status: 0=good, 1=positive overflow, 2=negative overflow, 3=undefined

Voltages above and below the 0-2.5V range can be measured: 10% over-range (+2.75V on the standard configuration) is permitted. Status=1 is returned above +2.5V. Negative voltages down to -0.15V can be measured, with -0.1V being the maximum negative value where accuracy is maintained; this is imposed by the -0.15V negative supply rail together with input protection diodes in IC3 and IC5. Negative overflow (status=2) will not appear because the aforementioned clamping prevents exceeding -2.5V.

No isolation is provided; the SMA connector ground is connected to the KDE485 digital ground.

Ethernet

Physical Layer Description

This is a standard 10/100 interface with galvanic isolation, auto-MDIX and auto cable insertion detection.

POE (Power Over Ethernet) is not supported. Do not connect the KDE485 to POE injectors which are non-standards-compliant as these can damage the interface.

Enabling Ethernet

Ethernet and the TCP/IP stack (LwIP) are globally enabled in config.ini with

```
eth=1
```

Ethernet is a large part of the KDE485 firmware, so if disabled, the 64k RTOS stacks area expands dramatically.

Note that eth=0 blocks all config access to the KDE485, except via USB.

Ethernet Basic Configuration

For test purposes, the default MAC and the MTU (default=1500) values can be overriden with e.g.

```
eth_mac=007fe3d2f401        ; bit 0 of byte 1 must be 0 (because 1=multicast)
eth_mtu=1300                ; max 1500
```

LwIP Ethernet API

Overview

The ethernet API is the widely used LwIP library, version 2.0.3. LwIP has 3 interfaces:

1) RAW API - not thread safe
2) Netconn API - simple to use (see KDE_http_server.c for usage examples)
3) LwIP Sockets - similar to the API used in the C/Unix/Posix/Linux world

The RAW API is useful for LwIP applications without an RTOS, so not especially useful on the KDE485.

The Netconn API is simpler than the Sockets API. Examples can be found in the source code of the KDE485 HTTP Server - KDE_http_server.c.

The Sockets API is widely used. Examples can be found in the source code of KDE_http_client.c.

LwIP is an open source library. More information can be found in many places online e.g. https://www.nongnu.org/lwip/2_0_x/raw_api.html.

Application Tips

LwIP has been proven in the field for over 15 years and is relatively simple to use for anyone familiar with BSD Sockets, or the LwIP-specific Netconn API on top of which the BSD Sockets API is built.

LwIP and RTOS task priority

FreeRTOS is pre-emptive, with a 1ms tick. Even if a task runs in a loop without yielding to RTOS, it will still get pre-empted by other tasks whose priority is the same or higher.

When an application makes an LwIP API call (Netconn or Sockets) the code initially runs in the calling application's task, and runs at that task's priority. At some point during the API call the code will attempt to acquire the LwIP core lock mutex, and if necessary, will block waiting for another task to release it. However, since the LwIP code is designed to never block while holding the core lock, the wait time will be small, and in many cases zero. Once the core lock is acquired, the LwIP code running on the application task completes the requested API operation, releases the core lock and returns to the application code. In this way, the LwIP core lock serves to serialize all activities in the LwIP stack, so as to protect shared data structures.

Normally this works well. However, if the system is busy, LwIP can run out of buffers. When this happens, the application task will block in the API call waiting for more buffers to be available. The LwIP TCP/IP task will then periodically check the state of the buffer pool, and once buffers are available, it will complete the outstanding API work and wake the application task.

When buffers are low, other tasks calling the LwIP API concurrently will not block unless they too need buffers to complete their work. Compute-bound tasks which are not calling LwIP (e.g. TLS performing heavy crypto) will not affect activity in LwIP regardless of buffer availability unless the priority of the compute-bound tasks is higher than that of the LwIP app tasks, or the LwIP TCP/IP task itself.

Therefore, for best communications performance, the LwIP TCP/IP task runs at a high priority (40), and the LwIP application tasks should run at a lower priority. Finally, compute-bound tasks which make minimal or no use of LwIP should run at a lower priority still.

Broadcasts

There is a risk in misconfigured networks with broadcast storms. The KDE485 discards all multicast packets, except ARP broadcasts which are supported because without them most functionality is lost. If eth_multi=1 in config.ini all multicast packets are passed through to LwIP. This can also be done dynamically with the global variable g_eth_multi=true.

IPV6

IPV6 is not supported by the current LwIP build in the KDE485.

Network Services

The following services are implemented via ethernet:

DHCP and IP configuration

This is implemented at startup, at any Ethernet cable insertion, and thereafter at each DHCP lease expiry. The following **config.ini** parameters, checked at startup, apply:

```
dhcp=1                      ; 1 = obtain IP via DHCP; 0 = disable DHCP
dhcp_retries=10             ; number of retries
ip_static=169.254.75.75     ; IP allocated if dhcp=0 or if DHCP fails
ip_mask=255.255.255.0       ; used with static IP
ip_gateway=169.254.75.75    ; used with static IP
ip_dns_server=8.8.8.8       ; used with static IP
```

If DHCP fails after dhcp_retries, the KDE uses the ip_static value above.

The actual IP in use is written into the boot.txt file. See **boot.txt** for details. Due to IP establishment time, it may not appear in boot.txt immediately after KDE startup.

In most industrial applications the KDE485 will be on a fixed IP and the gateway IP is needed:

```
dhcp=0
ip_static=192.168.1.55
ip_mask=255.255.255.0
ip_gateway=192.168.1.1      ; IP of the router
ip_dns_server=8.8.8.8       ; standard google-provided DNS
```

NTP Client

This sets the RTC (real time clock) at power-up and at specified intervals thereafter; this function is auto-disabled if a GPS is delivering valid time data. Your router or firewall needs to have port 123 open to the outside, plus port 53 (DNS) if the config.ini file contains a hostname instead of an IP. Most routers already allow all outbound traffic. The NTP Client is practically necessary for any KDE application which needs time, unless NTP Server (below) is being used, with GPS as the time source.

The NTP Client function is always running and, if a server hostname or IP is configured (the ntp_server parameter) is looking for the specified NTP server. The following config.ini parameters apply:

```
ntp_server=time.windows.com ; the NTP server hostname (or IP e.g. 40.119.148.38)
ntp_max_diff=10             ; max jump, in seconds (always +ve value)
ntp_jump_first=1            ; 1 allows unlimited jump on first update
ntp_resync=1                ; interval (hours) between checks
```

Normally, if a GPS is connected (see below) set ntp_server=0.0.0.0 (the address of the NTP server on the internet) to stop the NTP Client trying to get time from the internet.

NTP Server

This is intended for use where a GPS is the date/time source. It enables the KDE to act as a timeserver on a LAN which for security or other reasons has no connection to the internet. It serves NTP requests from an NTP client, fulfilling the request from the KDE485 RTC values. This GPS can be the internal SPI-GPS factory option, or an external NMEA GPS receiver connected via RS232 or RS422 (i.e. ports 1,2,4). The RTC is updated from GPS once a minute. A high quality GPS positioning connection is not needed to get date/time; a single satellite is enough. The GPS feature starts only if the gps_rtc parameter in config.ini is configured:

```
gps_rtc=0                   ; no GPS connected
gps_rtc=2,38400,0           ; GPS on port 2, 38400 baud, no GPS initialisation
gps_rtc=2,38400,1           ; as above, send U-BLOX NEO-M9N initialisation string
gps_rtc=-1                  ; as above, internal GPS factory option installed
```

The following example configuration will produce an NTP server on 192.168.1.200

```
dhcp=0
ip_static=192.168.1.200
ip_mask=255.255.255.0
ip_gateway=192.168.1.1
ip_dns_server=8.8.8.8
ntp_server=0.0.0.0
dhcp_retries=10
ntp_max_diff=10
ntp_jump_first=1
ntp_resync=24
gps_rtc=-1                  ; for SPI GPS and updating RTC from it
```

To prevent spurious time data being served, the NTP Server does not respond until a GPS time has been initially obtained after power-up, and the seconds have passed through 00.

Note that if a valid NTP server is configured (see **NTP Client** above) then, if GPS data is later lost, the KDE will attempt to get time from that NTP server, and it will serve this to the client(s) instead of GPS time. Often this is not desired, or external internet access is firewalled, in which case set ntp_server=0.0.0.0. The KDE will automatically revert to serving GPS time when GPS reception is restored. If neither GPS time nor NTP time is available, the KDE will serve time from its internal RTC (real time clock) which is based on a crystal that is accurate to around 30ppm.

HTTP Server

A simple HTTP server is provided for local and remote KDE485 management

HTTP Client

This exists via an API documented in KDE_http_client.h. This is similar to (but simpler than) the HTTPS Client API which is documented in KDE_https_client.h.

Example HTTP Client code can be found in KDE_healthchecks.c.

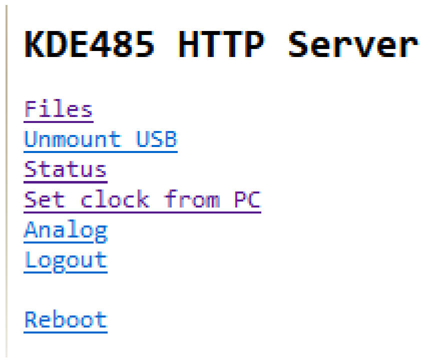
HTTPS Client

The HTTPS Client is similar to the HTTP Client, with additional API functions to handle the security aspects. It invokes the TLS module.

HTTP Server

Overview

This is a simple function intended for local (LAN) KDE485 management, config, status, diagnostics, etc.


KDE485 HTTP Server

Configuration

The server is configured with the following config.ini parameters:

```
http_svr=1                    ; 1 = enable HTTP server (default=1)
http_svr_name=kde485          ; login username - max length 20, min 1
http_svr_pwd=12345            ; login password - max length 20, min 1
http_svr_port=80              ; port - optional, max 65535, default 80
http_svr_logout=3600          ; auto logout - optional, seconds, default=3600, 0 disables logout
http_svr_client=0.0.0.0       ; mandatory client IP (enforced if not 0.0.0.0)
```

Going with a browser to **IP:port/kde485.html** (the IP can be found in boot.txt in the KDE485 filesystem, over USB) presents a web page. The :port portion is optional.

The login requires a username and a password, plus the optional client IP match.

The browser may complain that the site is insecure, etc. This is normal for modern browsers which increasingly do not like HTTP servers. Disregard this warning.

If you change the login credentials or the client IP (by editing config.ini) the changes are effective immediately.

The login is not user context sensitive (no cookies, etc). In the absence of client activity, the login status expires after http_svr_logout.

Various functions are available as shown above. These include a filesystem directory listing with edit, delete and upload functions, and a status page showing various RTOS tasks running and other system parameters and options.

Files

The file operations are based on the KDE485 FAT12 filesystem i.e. root-only (subdirectories not supported) and, 8.3 filenames. The Files display shows any subdirectories in the root (e.g. created from Windows via USB) and these can be deleted if they are empty. The File Upload function checks the selected filename is valid-8.3 and is not too big for the available space.

Filesystem access speed via the HTTP server is 1Mbytes/sec for reading and 30kbytes/sec (defined by the FLASH writing speed) for writing. Note that the FLASH write code does a pre-read, on each 512 byte sector, to check if the data being written is actually being changed. This can result in file writes running extremely fast!

The KDE485 will create a default config.ini at startup if it finds there is none, or the existing one is substantially corrupted, so the filesystem always remains accessible via USB but, for additional protection, if you edit config.ini this creates a backup config.bak.

Note that while changes in the KDE485 filesystem done by a) internal software or b) by the HTTP server functions are **immediately** visible to the other of these two, they are not usually detected by the USB host (due to Windows caching of removable drives), and an Unmount USB function has been provided on the main page for this purpose.

Note also that editing the config.ini file could affect other RTOS tasks if they read it periodically. Most read it only at startup but there are exceptions.

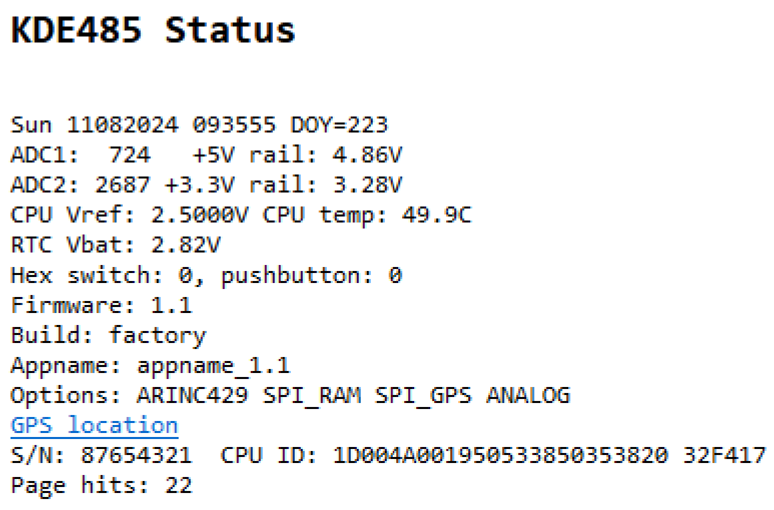
The Edit function is enabled only for .txt .ini .cfg files. Following the KDE485 config.ini convention, the edited file is saved with CRLF line endings even if it previously had LF (unix style) line endings.

Unmount USB

This causes the USB Host to re-read the file system. Most operating systems treat removable media as their sole property and do not periodically re-scan it

Status

This page shows a variety of information about the USB485, including the current status of all running RTOS tasks.



KDE485 Status

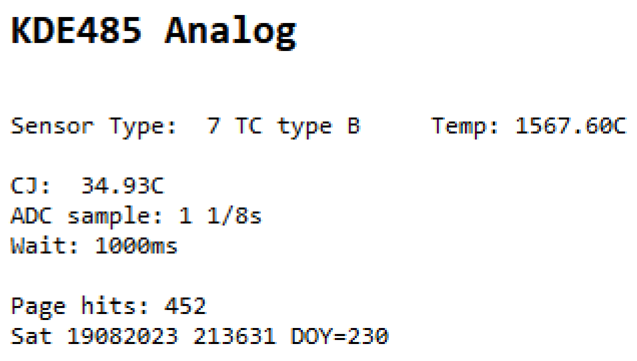
```
Sun 11082024 093555 DOY=223
ADC1:  724   +5V rail: 4.86V
ADC2: 2687 +3.3V rail: 3.28V
CPU Vref: 2.5000V CPU temp: 49.9C
RTC Vbat: 2.82V
Hex switch: 0, pushbutton: 0
Firmware: 1.1
Build: factory
Appname: appname_1.1
Options: ARINC429 SPI_RAM SPI_GPS ANALOG
GPS location
S/N: 87654321  CPU ID: 1D004A001950533850353820 32F417
Page hits: 22
```

Set clock from PC

This sets the USB485 RTC from the client PC time in UTC. Note that this will be overriden shortly afterwards if the KDE485 is getting time sync from NTP or GPS.

Analog

This displays the value read by the optional isolated analog subsystem. The Sensor Type and other values reflect the settings selected in the config.ini file.



KDE485 Analog

```
Sensor Type:  7 TC type B     Temp: 1567.60C

CJ:  34.93C
ADC sample: 1 1/8s
Wait: 1000ms

Page hits: 452
Sat 19082023 213631 DOY=230
```

The data can be extracted from the HTTP page in an automated matter with utilities such as wget which are available for both Unix and Windows:

```
wget http://192.168.1.63/analog.html
```

which creates a file analog.html containing the data. This file can be parsed with grep, etc. to extract the value and the timestamp.

The Analog page auto refreshes at approx 1Hz, using HTML in the client browser. Note that there could be inadvertent synchronisation between this and the frequency at which the data is updated, which could result in the data appearing static.

Logout

This sets the HTTP server back into the non logged-in state.

Reboot

This reboots the KDE485

Multiple Clients

The HTTP server does not implement sessions or cookies so can be accessed by multiple clients concurrently, limited only by the the http_svr_client IP limitation, if any. This works because the response to HTTP requests is practically immediate.

Security

In line with standard "IOT" security practices, an "embedded product", particularly one with an obvious server functionality, should not be exposed to the open internet (e.g. by opening up port 80 in your NAT router or firewall). The main reasons are

- in today's hacking climate, only a proper heavy duty server should be exposed to the internet
- some form of a firewall is mandatory; this can be NAT (with no open ports) or a real firewall
- embedded firmware is rarely if ever updated
- industrial networks should not have a direct internet connection
- physical access control is rarely implemented

Whether HTTP or TLS/HTTPS is used makes no difference to the above.

On the plus side, the KDE485 is very difficult to attack productively because it isn't a "normal computer". For example, short of building an application using the Cube IDE kit and loading it via USB (which needs physical access) or the HTTP server (which can be disabled with http_svr=0 in config.ini), there is no way to load an executable program into the filesystem and get it to run. This is not linux, and the required API is simply not there. There is also no scripting language. Using the KDE485 as a proxy to attack other devices would be extremely hard because it is not running any recognisable operating system. In modern terminology the "attack surface" is negligible for remote attacks.

DOS (denial of service) is possible. The KDE485 minimises this with an Ethernet implementation which uses RTOS-polled - not interrupt-driven - reception, optimised by a variable polling interval, and by discarding non-ARP multicast packets (eth_multi=0).

If you need remote access to the KDE485 HTTP server, use a VPN, or RDP (Remote Desktop) over a VPN, etc, to connect to your LAN securely. A VPN usually terminates on a fixed LAN IP (which depending on the VPN type can be an IP on the KDE's LAN, or an IP on the subnet of the remote machine) and the http_svr_client feature can be used to lock the KDE485 HTTP server login to that VPN IP. Note that the IP is checked only at the login.

TLS

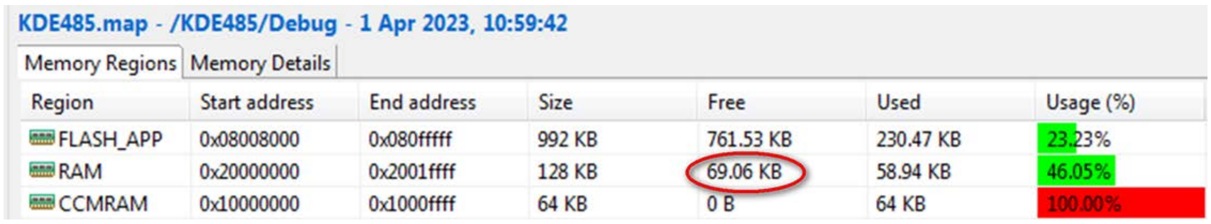
Overview

Mbed TLS is a C library that implements cryptographic primitives, X.509 certificate manipulation and the SSL/TLS and DTLS protocols. The main documentation can be found here https://mbed-tls.readthedocs.io/en/latest/

TLS is implemented using the open source MbedTLS library v 2.16.2 and is supplied in source form. It is a standalone module in the KDE485 which is invoked by other RTOS tasks on demand.

Memory Requirements

MbedTLS runs within the invoking task's workspace and RTOS priority. It uses 48k of memory, which is allocated on the general heap when TLS is invoked and deallocated afterwards. Sufficient free RAM must be available for this 48k buffer to be allocated. Here we have 69k:



Region	Start address	End address	Size	Free	Used	Usage (%)
FLASH_APP	0x08008000	0x080fffff	992 KB	761.53 KB	230.47 KB	23.23%
RAM	0x20000000	0x2001ffff	128 KB	69.06 KB	58.94 KB	46.05%
CCMRAM	0x10000000	0x1000ffff	64 KB	0 B	64 KB	100.00%

TLS is used by the following modules which also serve as examples for user applications using HTTPS/TLS:

- HTTPS Client
- Healthcheck (if the URL is https://)
- Dropbox
- Auto update of cacert.pem certificate file

TLS and Multiple RTOS tasks

MbedTLS itself is thread-safe but there is not enough RAM available to invoke multiple TLS sessions so there is a mutex at top level which controls access to it. This means that if one RTOS task is using any part of TLS, another task cannot use TLS during that time. One exception to the foregoing is if a task is doing encryption internally (without invoking TLS) e.g. running a shared-key AES256 point to point data link using AES_CBC_encrypt_buffer(). See AES Encryption.

Certificate Chain Processing

The original open source MbedTLS code cannot process a string of certificates out of a file. It can process just one. This is OK if connecting to a private server which has a single (usually self-signed) certificate. The KDE485 implementation can process the whole root certificate store out of a filesystem file cacert.pem. The size of this file is limited only by the filesystem space; the current file is around 220k.

Auto Update of CACERT.PEM Certificate File

This is an RTOS task in KDE_cert_update.c. It stores the last modified timestamp in a file named CACERT.TXT. On completion, if the file has been updated, it will request a remount of the USB mass storage to prevent Windows from using its cached concept of the directory/FAT.

The new file is downloaded into a temporary file download.tmp and the filespace needs to have sufficient room for this (about 220k bytes currently). This temporary file is then deleted.

This auto update feature is activated using these config.ini values:

```
cert_update=0                                   ; set to 1 to enable automatic update
cert_update_url=https://curl.se/ca/cacert.pem   ; URL
cert_update_interval=24                         ; check interval in hours
```

Alternate Names

Subject alt names are not currently supported in TLS certificates and consequently TLS certificate verification will fail for sites for which the subject name in the certificate does not match the domain name.

One option is to relax the certification verification option in this situation although consideration should always be made of the security implications of this step.

TLS Active Indication

TLS uses significant resources. One of these is that the KDE485 filesystem is locked during the certificate file processing. A global flag bool g_TLS_active_FS is set during this time (see KDE_https_client.c) and this flag can be used to indicate to user code that e.g. ethernet may not be available. This flag is active for at most a few seconds.

Supported TLS Cipher Suites

Standard TLS 1.2 functionality is provided. All cipher suites specified by TLS 1.2 are supported. Most of the possible algorithms and possible ciphersuite combinations have been included along with most possible extensions.

TLS 1.0 and 1.1 are disabled, but can be re-enabled if required by editing mbedtls_config.h to uncomment the following options and recompiling:
MBEDTLS_SSL_PROTO_TLS1_0
MBEDTLS_SSL_PROTO_TLS1_1

A number of "deprecated" algorithms are included as they are still widely used, especially in the industrial field. For certificate chain verification, some of the root certificates are very old and use deprecated algorithms such as SHA1 (e.g. the Digicert Root certificate as used as the root CA by Dropbox, among others).

TLS 1.2 removes the use of MD5/SHA1 for signature verification, but not necessarily for key exchange purposes. TLS 1.3 is not yet supported by the KDE485 version of MbedTLS.

A Windows executable version of TLS is provided for convenient debugging of host issues.

Windows TLS Executable

Overview

A Windows command line executable program HTTPSClient.exe is provided, built using the same configuration options and version of mbedTLS that is used in the KDE485. This provides an easy way to test and troubleshoot HTTPS client connections.

While emulating the memory constraints and configuration of the embedded KDE485 version of the code, the Windows version is able to produce memory usage information and greater levels of debug information including a textual interpretation of the last error code (usually the reason that the session negotiation with the server failed.

Unlike the KDE485 version, the Windows version only currently allows for an HTTPS GET request - it does not allow the sending of larger amounts of data to the server. The primary intention of the Windows version is to be able to identify issues with the TLS negotiation. Once the TLS negotiation has succeeded, it is not normally an issue sending or receiving any quantity of data.

HTTPSClient.exe expects to find a cacert.pem file containing the root CA certificates in the same folder as the HTTPSClient.exe executable file. A suitable, up to date cacert.pem can be downloaded here:
https://curl.se/docs/caextract.html

Usage

Running HTTPSClient.exe without any arguments will output the following usage information:

```
KK Systems KDE485 HTTPS client test program
HTTPSClient [flags] [URL]
Optional flags are:
  -c      list ciphersuites
  -d      <LEVEL>     mbedTLS debug output level
  -h      HTTPS certificate verification debug output
  -r      memory usage report
  -m      <#BYTES>     memory size to allocate in bytes (default is 48KB)
  -v      certificate verification: 0 = off, 1 = optional,  2 = required(default)

Expects root certificate file cacert.pem to be found in the same directory as this executable
```

The minimum required to test HTTPS negotiation with a server is a URL (including the “https://”). For example, to negotiate a session and request the page https://www.kksystems.com enter:

```
HTTPSClient https://www.kksystems.com
```

This will output the minimum amount of information about the negotiation and the resulting page data (or a text description of any error encountered.

Additional TLS negotiation debug information options

Adding the flags -d, -h or -r will increase the amount of debug output during the session negotiation. For example, the following will output debug information from mbedTLS at level 3, will provide the details of each certificate verified in the root CA chain and show the chosen ciphersuite resulting from the negotiation. It will also show the amount of memory currently used as well as the maximum amount used at each step of the negotiation process.

```
HTTPSClient -d 3 -h -r https://www.kksystems.com
```

The -d option corresponds to the KDE g_debug_tls variable (or config.ini option debug_tls) which sets the mbedTLS internal debug log level. This option expects the next argument to be the integer value to use for the debug log level.

The -h option corresponds to the KDE g_debug_https variable (or config.ini option debug_https) which enables output of the root CA certificate chain information and the chosen ciphersuite.

The -r option shows the amount of mbedTLS heap memory allocated at each step. There is currently no equivalent of this option on the KDE itself.
Other options
The -c option will list all of the enabled ciphersuites including their name and 16 bit identification value (as used in the TLS negotiation). This option can be used with or without a URL. For example, the following will just list the ciphersuites without performing a TLS negotiation or HTTPS request:

```
HTTPSClient -c
```

The -v option corresponds to the KDE g_https_verify variable (or config.ini option https_verify). It controls whether the HTTPS client should verify the server's certificate. It expects an integer value as the next command line argument. Values are 0 = off, 1 = optional or 2 = required (default).

The -m option allows the size of the heap made available to mbedTLS to be specified. The default value if this option is not specified is 48KB (49152 bytes) as it is in the KDE. The -m option expects an integer number of bytes to be specified as the next command line argument. Problems with TLS negotiation with some servers or some ciphersuites may be resolved in future by increasing the amount of heap memory available to mbedTLS and this option allows it to be easily tested or verified that any negotiation problem is due to lack of memory.

For example, the following example specifies a heap of 52KB (53248 bytes) should be used with the page request to www.kksystems.com:

```
HTTPSClient -m 53248 https://www.kksystems.com
```

AES Encryption

TLS is an overkill for many secure datacomms applications. Certificate management and particularly expiration is a huge problem in all of IT. In embedded products, given the absence of easy PC-like user interfaces, the management of certificates can be a challenge and if this is not done correctly the system will suddenly stop working and fixing certificates, especially years after initial commissioning, is likely to be problematic.

If both ends of the link are regarded as physically secure then a simple shared key system works perfectly. It also provides automatic mutual authentication, man-in-the-middle attack protection, etc, because a third party cannot participate in the communication without the key.

The KDE485 offers two ways to access encryption:

Using AES256 from "Tiny AES"

This runs at 100kbytes/sec. It is a very compact (under 1k) software implementation and is recommended for all applications where the speed is sufficient. It is thread-safe. The following functions are provided

```
void AES_init_ctx_iv(struct AES_ctx* ctx, const uint8_t* key, const uint8_t* iv);
void AES_CBC_encrypt_buffer(struct AES_ctx* ctx, uint8_t* buf, uint32_t length);
void AES_CBC_decrypt_buffer(struct AES_ctx* ctx, uint8_t* buf, uint32_t length);
```

There are many variants of "Tiny AES" and the one in the KDE485 is based on https://github.com/kokke/tiny-AES-c/tree/master.

Using AES256 from the TLS Suite

In the original MbedTLS **software** version this runs at 800kbytes/sec (for AES256). In the KDE485 it uses the 32F417's **hardware** AES acceleration and can be AES128 AES192 or AES256. With the AES hardware acceleration it runs considerably faster than the KDE485 ETH interface so in general there isn't much point in using it over the Tiny AES version. However, below is information on how to access it:

For the full mbedTLS AES API, see the include file:
Middlewares\Third_Party\mbedTLS\include\mbedtls\aes.h

Initialising AES

Your application will need to allocate (on the heap, stack or in the data segment) an mbedtls_aes_context which is used to maintain the key schedule and state of any of the AES modes.

The following functions initialise and release the supplied AES context structure.

```
void mbedtls_aes_init(mbedtls_aes_context *ctx);
void mbedtls_aes_free(mbedtls_aes_context *ctx);
```

Setting a key

The AES context maintains separate keys for encryption and decryption

```
// keybits indicates the AES key length and may be 128, 192 or 256
int mbedtls_aes_setkey_enc(mbedtls_aes_context *ctx,
const unsigned char *key,
unsigned int keybits);

int mbedtls_aes_setkey_dec(mbedtls_aes_context *ctx,
const unsigned char *key,
unsigned int keybits);
```

Encrypting and decrypting

```
// In each case below, the mode param may be MBEDTLS_AES_ENCRYPT or
MBEDTLS_AES_DECRYPT as required.

// Use AES-ECB (electronic code book) mode
int mbedtls_aes_crypt_ecb(mbedtls_aes_context *ctx,
                          int mode,
                          const unsigned char input[16],
                          unsigned char output[16]);

// Use AES-CBC (cipher block chain) mode
int mbedtls_aes_crypt_cbc(mbedtls_aes_context *ctx,
                          int mode,
                          size_t length,
                          unsigned char iv[16],
                          const unsigned char *input,
                          unsigned char *output);

// Use AES-CFB128 (128 bit cipher feedback) mode
int mbedtls_aes_crypt_cfb128(mbedtls_aes_context *ctx,
                          int mode,
                          size_t length,
                          size_t *iv_off,
                          unsigned char iv[16],
                          const unsigned char *input,
                          unsigned char *output);

// Use AES-CFB8 (8-bit cipher feedback) mode
int mbedtls_aes_crypt_cfb8(mbedtls_aes_context *ctx,
                          int mode,
                          size_t length,
                          unsigned char iv[16],
                          const unsigned char *input,
                          unsigned char *output);

// Use AES-OFB (output feedback) mode
int mbedtls_aes_crypt_ofb(mbedtls_aes_context *ctx,
                          size_t length,
                          size_t *iv_off,
                          unsigned char iv[16],
                          const unsigned char *input,
                          unsigned char *output);

// Use AES-CTR (counter) mode
int mbedtls_aes_crypt_ctr(mbedtls_aes_context *ctx,
                          size_t length,
                          size_t *nc_off,
                          unsigned char nonce_counter[16],
                          unsigned char stream_block[16],
                          const unsigned char *input,
                          unsigned char *output);
```

Example

Note that the following example disregards error handling for the benefit of clarity:

```
{
mbed_tls_aes_context aes_context;
unsigned char key[16] = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
                          0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70, 0x80 };
unsigned char iv[16] = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
                          0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

char plaintext[]  = “abcdefghijklmnop”;
unsigned char ciphertext[16];

memset(ciphertext, 0, sizeof(ciphertext));

// Initialise the AES context
mbedtls_aes_init(&aes_context);

// Set an 128 bit AES key
mbedtls_aes_setkey_enc(&aes_context, key, 128);

// Use CBC mode to encrypt the data
mbedtls_aes_crypt_cbc(&aes_context,
                          MBEDTLS_AES_ENCRYPT ,
                          sizeof(ciphertext),
                          iv,
                          plaintext,
                          ciphertext);

// Release the context
mbedtls_aes_free(&aes_context);
}
```

Other Algorithms

The TLS suite contains many other algorithms including the older ones like DES and 3DES. Any of these are just fine for typical industrial control applications (despite internet hype, none can be cracked in any realistic scenario) but using anything other than AES is pointless. Also, in the TLS suite, only AES uses the 32F417 hardware crypto. The 32F417 contains hardware acceleration for DES and 3DES but since almost nobody uses these today, the DES/3DES hardware acceleration was not implemented in TLS.

HTTPS Client

This is similar to the HTTP Client, with additional API functions to handle the security aspects.

The KDE485 does not (in the current hardware version) have enough memory to run an HTTPS Server and an HTTPS Client concurrently, particularly as a Server potentially needs to support multiple sessions. This limitation is due to the TLS memory requirements - of the order of 50k per session. Only the Client mode is supported; this also fits much better with the general IOT security requirement (see **HTTP Server** above).

The HTTPS client is implemented with MbedTLS. It is intended for connecting to a server on a LAN or on the internet.

Usage

The main method for accessing the server is via user code (see **HTTPS Client API Summary**) , although certain KDE485 services (e.g. healthcheck) also use it.

The following config.ini values are applicable:

```
debug_usb=0              ; 1 enables debugs with debug_thread_* functions
debug_https=0            ; 1 enables debugs from https/tls crypto operations
debug_tls=0              ; 1/2/3 = MbedTLS internal debug level (requires
                           debug_usb=1 also)
https_verify=2           ; 0/1/2 = MbedTLS client-server X509 verification
                           0=not used  1=optional  2=mandatory
```

For specific debugging of the HTTPS certificate chain parsing, and with all other debugs suppressed, use

```
debug_usb=0
debug_https=1            ; this overrides debug_usb=0
```

Certificates

An HTTPS client does not need to hold a pre-generated public+private key pair. To protect against DNS and server spoofing, an optional certificate of the server can be provided in the KDE filesystem in **CACERT.PEM**. This file can hold either a single certificate (for accessing a private server which has a single self-signed certificate) or it can contain the whole root CA certificate chain (for accessing a public server e.g. dropbox.com whose certificate needs to be verified by parsing a CA chain). For the latter case, the file will typically contain around 120 root certificates, some of which expire every few years while others last for 10-20 years. You need an admin process to keep this file up to date, and this is another reason why "IOT" products (products with internet capability) should be **clients** and these should access only servers configured with current security procedures.

If the file cacert.pem is missing, or is out of date, https_verify needs to be 0 or 1 to suppress server authentication.

The file cacert.pem is derived from Mozilla's list of root certificates and can be downloaded (for example) here:

https://curl.se/docs/caextract.html

The filename is defined in KDE_https_client.c as KDE_HTTPS_CA_ROOT_CERT_FILENAME.

The cacert.pem file can be automatically updated as described here.

The HTTPS Client runs in the RTOS task of the invoking function. It relies on the TCP/IP RTOS task running; this is started at power-up if eth=1 in config.ini.
The API below is also documented in KDE_https_client.h.
Example HTTPS Client code can be found in the KDE_dropbox and KDE_healthchecks sources.

HTTPS Client API Summary

In its simplest form, the HTTPS client can be used to send a single short HTTPS request/response with the following sequence:

```
KDE_https_client_init();
KDE_https_client_simple_request();
KDE_https_client_close();
```

Larger amounts of data may be sent/received as part of a single request with a number of separate function calls to start a request and then supply any number of blocks of data in a sequence of calls, read the response data in blocks and then finish the request. For example, the following sequence can be used:

```
KDE_https_client_init();
...
KDE_https_client_start_request();
KDE_https_client_send_data();
KDE_https_client_send_data();
...
KDE_https_client_receive_data();
KDE_https_client_receive_data();
...
KDE_https_client_end_request();
...
KDE_https_client_close();
```

Another form of the simple_request function allows a single function to be used to send an arbitrary length request. A callback function can be used to supply the data in as many blocks as required to achieve the specified content length:

```
KDE_https_client_init();
```

This will call the supplied callback function as many times as required to fetch the data

```
KDE_https_client_simple_request_ex();
KDE_https_client_close();
```

HTTPS Client API Details

```
void KDE_https_client_init(
      uint8_t *memory,    // In  Pointer to block of contiguous memory
      size_t length       // In  Size of memory block in bytes
      );
```

The HTTPS client must first be initialised by providing a large block of working memory. Currently at least 48Kbytes of memory is required. This may be provided by doing a single malloc or by using a static array.

This function must be called before any other HTTPS functions but may be safely called repeatedly (it will have no effect if called again) before KDE_https_client_close().

```
void KDE_https_client_close(void);
```

This function de-initialises the HTTPS client, releasing the memory supplied by KDE_https_client_init().

```
bool KDE_https_simple_request(
      char *server,       // In  Name of server to send request to
      char *port,         // In  Port number on server (decimal number as string)
      char *request,      // In  Full request content
                             (including GET/POST and header data)
      char *response,     // Out Where to put the response
      size_t max_resp_len,// In Maximum length of data to write to response
      uint32_t *resp_len  // OutOptional actual length of response
      );
```

Send a request and receive a short response in a single call without additional data using HTTPS. Note that KDE_https_client_init() must be called before this function is used.

This function returns true if successful.

```
bool KDE_https_simple_request_ex(
      char *server,       // In  Name of server to send request to
      char *port,         // In  Port number on server (decimal number as string)
      char *request,      // In  Request content (including GET/POST and header
                             data)
      uint8_t *content,   // In  Additional request content (may be NULL)
      KDE_TLS_DATA_CALLBACK callback,  // In  Callback function to provide content
                                          data (may be NULL)
      uint32_t content_length,         // In  Length of content in bytes
                                          (will use content or callback)
      char *response,     // Out Where to put the response
      size_t max_resp_len,// In Maximum length of data to write to response
      uint32_t *resp_len, // OutOptional actual length of response
      int32_t *result     // Out Optional result code (may be NULL)
      );
```

This is an extended version of the simple request function to send a request and receive a response in a single call using HTTPS. KDE_https_client_init() must be called before this function is used. Additional request content (e.g POST data) may be provided as a single data buffer or a callback function may be provided to return the content in arbitrary sized chunks.

This function returns true if successful. In the event of failure, an additional result code may be optionally returned via the result output parameter.

If a callback function is used this will be called until all of the previously specified content_length has been supplied. The callback function has the following form:

Params: data is a returned pointer to a block of data
Returns: true if success, false for failure

```
typedef bool (*KDE_TLS_DATA_CALLBACK)(
      uint8_t **data,           // Out Receives the address of the data
      uint32_t *length          // Out The length of the supplied data in bytes
      );
```

```
bool KDE_https_start_request(
      char *server,             // In  Name of server to send request to
      char *port,               // In  Port number on server
                                   (decimal number as string)
      char *request,            // In  Request content
                                   (including GET/POST and header data)
      KDE_TLS_HANDLE **handle,  // Out The handle to allocated resources for this
                                   session
      int32_t *result           // Out Optional result code (may be NULL)
      );
```

This function starts the process of sending a request and receiving a response in multiple calls using HTTPS. KDE_https_client_init() must be called before this function is used. The content_length of the data to be sent must be known when this function is called.

This function returns true if successful. In the event of failure, an additional result code may be optionally returned via the result output parameter.

```
bool KDE_https_send_data(
      KDE_TLS_HANDLE *handle,   // In  Handle for TLS session returned by
                                   KDE_https_start_request()
      uint8_t *data,            // In  Data to send
      uint32_t length,          // In  Length of data supplied in bytes
      int32_t *result           // Out Optional result code (may be NULL)
      );
```

Optionally call this function (as many times as necessary) to send additional data (e.g. POST data) for a request started with KDE_https_start_request(). The total data length should not exceed the content_length originally specified.

This function returns true if successful. In the event of failure, an additional result code may be optionally returned via the result output parameter.

```
bool KDE_https_receive_data(
      KDE_TLS_HANDLE *handle,   // In  Handle for TLS session returned by
                                   KDE_https_start_request()
      uint8_t *data,            // In  Where to write received data
      uint32_t *length,         // I/O In = max length to receive,
                                   Out = actual length received
      int32_t *result           // Out Optional result code (may be NULL)
);
```

Optionally call this function to read the response of a request started with KDE_https_start_request(). It will return a length of zero if no more data to receive (or actual length < max length indicates the end of data available).

This function returns true if successful. In the event of failure, an additional result code may be optionally returned via the result output parameter.

```
bool KDE_https_end_request(
      KDE_TLS_HANDLE *handle,   // In  Handle for TLS session returned by
                                   KDE_https_start_request()
      int32_t *result           // Out Optional result code (may be NULL)
);
```

This function ends a request that was previously started with KDE_https_start_request(). This will invalidate the supplied handle. It frees the memory associated with the session.

This function returns true if successful. In the event of failure, an additional result code may be optionally returned via the result output parameter.

HTTPS Server

An HTTPS Server has been experimentally implemented in the KDE485 but is not present for a number of reasons.

It uses around 60k of RAM and there is insufficient memory to run it together with an HTTPS client (which is much more useful). And each client session requires as much memory again.

Unless one has control over the client(s), the server needs to provide maximum size buffers (2x16k) whereas a client can reduce the TX buffer according to what it is actually transmitting.

There should be very few applications where an "IOT" device should be presenting a public-facing server (HTTP or HTTPS) because doing so requires the KDE485 to be on an open port, which exposes any vulnerability to anybody who wants to hack it.

The server uses a self-signed test certificate which causes modern browsers to display warnings before allowing the connection, and this browser issue is becoming worse over time. So a "proper" individual certificate would have to be purchased for each KDE485 server, which is impractical, unless all of them are on the same subdomain in which case a single certificate would do. The only way around this is if all clients are machines designed to accept self signed certificates.

Keep Alive

This feature is intended for applications involving 3G/4G routers and similar uses where the connection must be kept open. It transmits an ICMP (ping) packet to the specified host or IP. No response is expected. The following config.ini parameters apply:

```
kal=1                       ; enable keep-alive
kal_interval=60             ; keepalive period in seconds
kal_target=kksystems.com    ; keepalive target (hostname or IP)
```

Source code is provided so other forms can be implemented.

This function is disabled with kal=0 in config.ini.

Health Check

This function performs an HTTP or HTTPS access to an external site which registers whether the KDE is alive or not. There is a number of these on the internet (mostly free for up to a certain number of devices) or you can run one on your own server. They offer a web page showing the status of the various devices, and you can get email or telegram notifications of status, failures, etc.

The KDE function is set up for https://healthchecks.io. The following config.ini parameters apply (the URL is a non-functioning example):

```
healthcheck=1                   ; enable health check
healthcheck_interval=60         ; interval in seconds
healthcheck_url=https://hc-ping.com/8cd88cd4-ec2b-4a44-b5d2-d5f39ad124b7
```

This function is disabled with healthcheck=0 or with healthcheck_interval=0.

The URL is obtained by registering with healthchecks.io and is normally specific to each device. Multiple devices can share the same one, but then one cannot tell which one is running or has stopped running.

The mode (HTTP or HTTPS) is determined from the above URL. If HTTPS is specified, the KDE uses a TLS Client mode. The HTTP mode is much faster, and does not interfere with HTTPS Client operations.

Source code is provided so other forms can be implemented.

Dropbox

The KDE Dropbox API uses the KDE HTTPS Client to implement the Dropbox v2 API for transferring data from the KDE to a file within your Dropbox account. The Dropbox API 2 call used for this purpose is:

https://content.dropboxapi.com/2/files/upload

This API provides two ways of uploading file data to Dropbox:

A simple, single function call for creating files containing small amounts of data is KDE_dropbox_upload_file().

Alternatively, if you want to transfer arbitrary length data to a file, you can use the following sequence:

```
KDE_dropbox_upload_start()
KDE_dropbox_upload_send_data()
KDE_dropbox_upload_send_data()
.....
KDE_dropbox_upload_end()
```

In either case, the total length of data to be written should be known before the transfer is started.

Setting up your Dropbox "app"

To allow uploading to Dropbox via this API you will need to register a "Dropbox API App" on your Dropbox account. A Dropbox API app is simply a way of associating a number of permissions and authentication information with a folder inside your Dropbox account. This KDE Dropbox API will use the Dropbox App that you specify to transfer data to your account.

You can register a Dropbox App by going to this address:

https://www.dropbox.com/developers/apps/create

Select the options to register an app with "scoped access" for use with the Dropbox for HTTP API (the wording and user interface layout for these options changes so may not be exactly as described here in future).

To create files in an existing shared folder, you will need to register for "Full Dropbox" access which will give your "app" access to your entire Dropbox. Otherwise, if you select "App Folder", the API will only allow you to create files in a new folder that will be specifically created in the /Apps folder of your Dropbox account.

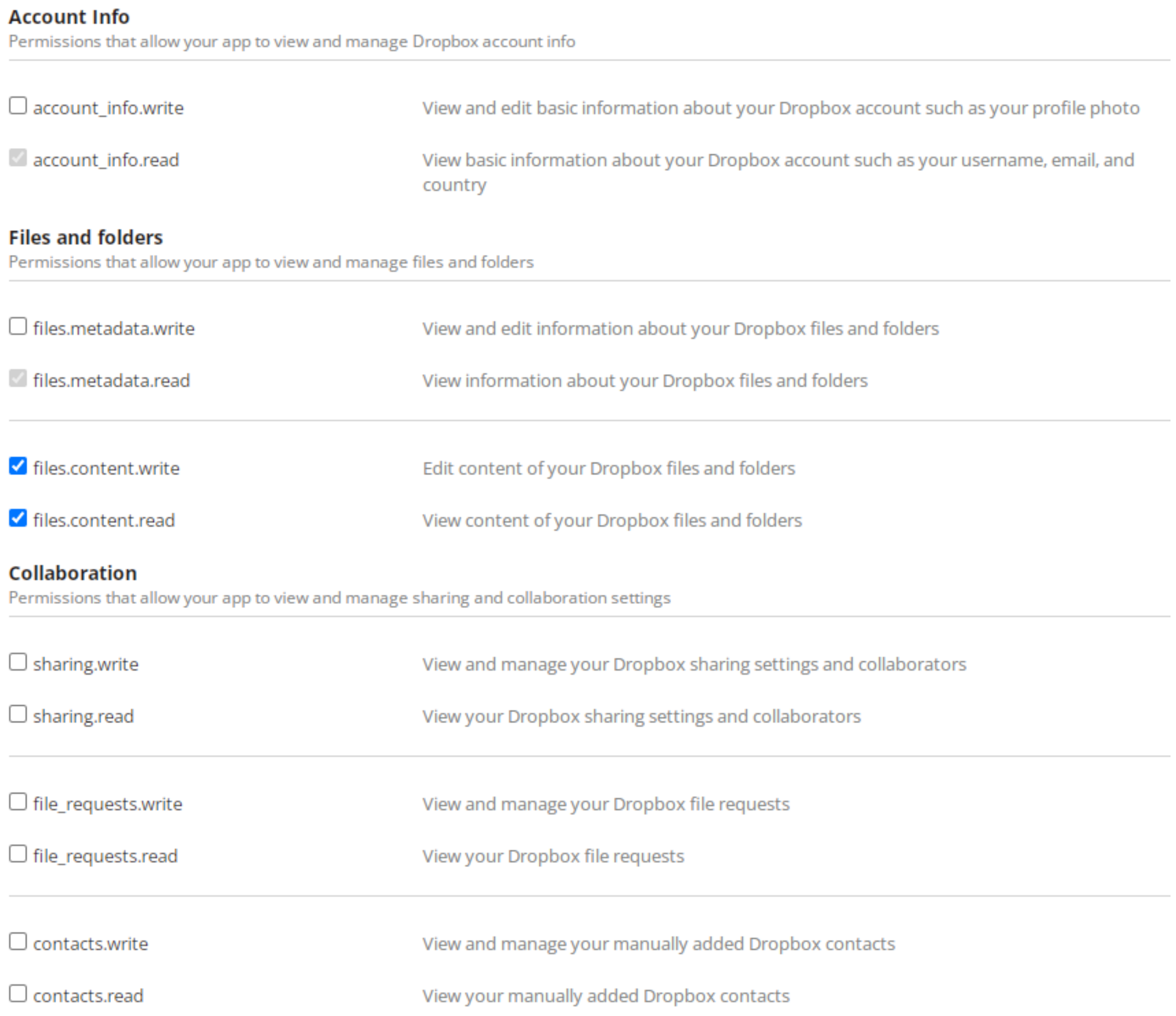
Choose a suitable name for your Dropbox App and set any other options as required.

Once you have created the app you will be sent to the Dropbox App Console "My Apps" page. Here you should select your new App and find the Permissions tab.

You can go directly to the App Console by entering the following URL (and logging in to your Dropbox account if necessary):

https://www.dropbox.com/developers/apps

Ensure that the permissions include files.content.write and files.content.read



Account Info
Permissions that allow your app to view and manage Dropbox account info

☐	account_info.write	View and edit basic information about your Dropbox account such as your profile photo
✓	account_info.read	View basic information about your Dropbox account such as your username, email, and country

Files and folders
Permissions that allow your app to view and manage files and folders

☐	files.metadata.write	View and edit information about your Dropbox files and folders
✓	files.metadata.read	View information about your Dropbox files and folders
✓	files.content.write	Edit content of your Dropbox files and folders
✓	files.content.read	View content of your Dropbox files and folders

Collaboration
Permissions that allow your app to view and manage sharing and collaboration settings

☐	sharing.write	View and manage your Dropbox sharing settings and collaborators
☐	sharing.read	View your Dropbox sharing settings and collaborators
☐	file_requests.write	View and manage your Dropbox file requests
☐	file_requests.read	View your Dropbox file requests
☐	contacts.write	View and manage your manually added Dropbox contacts
☐	contacts.read	View your manually added Dropbox contacts

The Dropbox API has changed recently (late 2021) and now only allows newly created apps to be used with "short-term" access tokens. The KDE Dropbox API provides the additional logic required to automatically request and manage the short-term access tokens but some extra initial steps and configuration is required to achieve this.

Go to the Dropbox App Console (https://www.dropbox.com/developers/apps) for your account and find the App Key for the Dropbox App that you have created. For example:

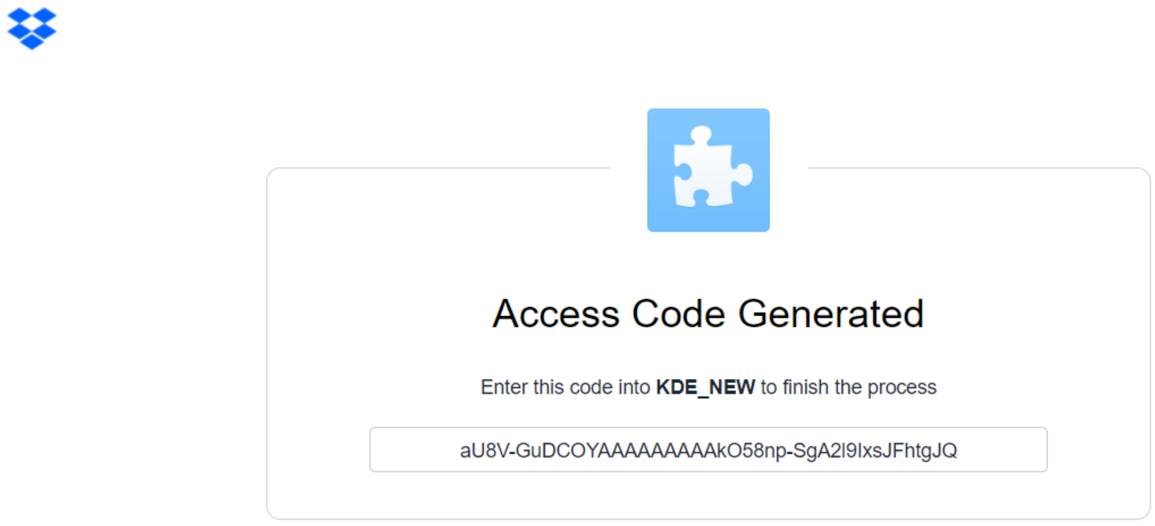
Next, enter the following URL in a web browser in order to obtain a one-time auth code (also known as an access-token). Modify this URL to include the App Key for your own app as shown in the Dropbox app console.

https://www.dropbox.com/oauth2/authorize?client_id=<INSERT YOUR APP KEY HERE>&token_access_type=offline&response_type=code

For example:

https://www.dropbox.com/oauth2/authorize?client_id=fhdq6ls9utqksem&token_access_type=offline&response_type=code

Follow the web site prompts to allow access for your app and you should be given an access code. For example:


Access Code Generated

Enter this code into **KDE_NEW** to finish the process

aU8V-GuDCOYAAAAAAAAAkO58np-SgA2l9IxsJFhtgJQ

Copy and paste this access code into the KDE config.ini file along with the app key and app secret for your app from the app console. For example:

```
dropbox=1       ; enable dropbox demo application RTOS task
dropbox_auth_code=aU8V-GuDCOYAAAAAAAAAkO58np-SgA2l9IxsJFhtgJQ
dropbox_app_key=fhdq6ls9utqksem
dropbox_app_secret=q17zs0grq0g1gjf
```

For demo/test purposes with the example code, add the following values which will cause a new file to be written to your Dropbox app folder every 120 seconds. A unique filename will be created for each upload by appending the date/time (for example data20220326-140356.txt)

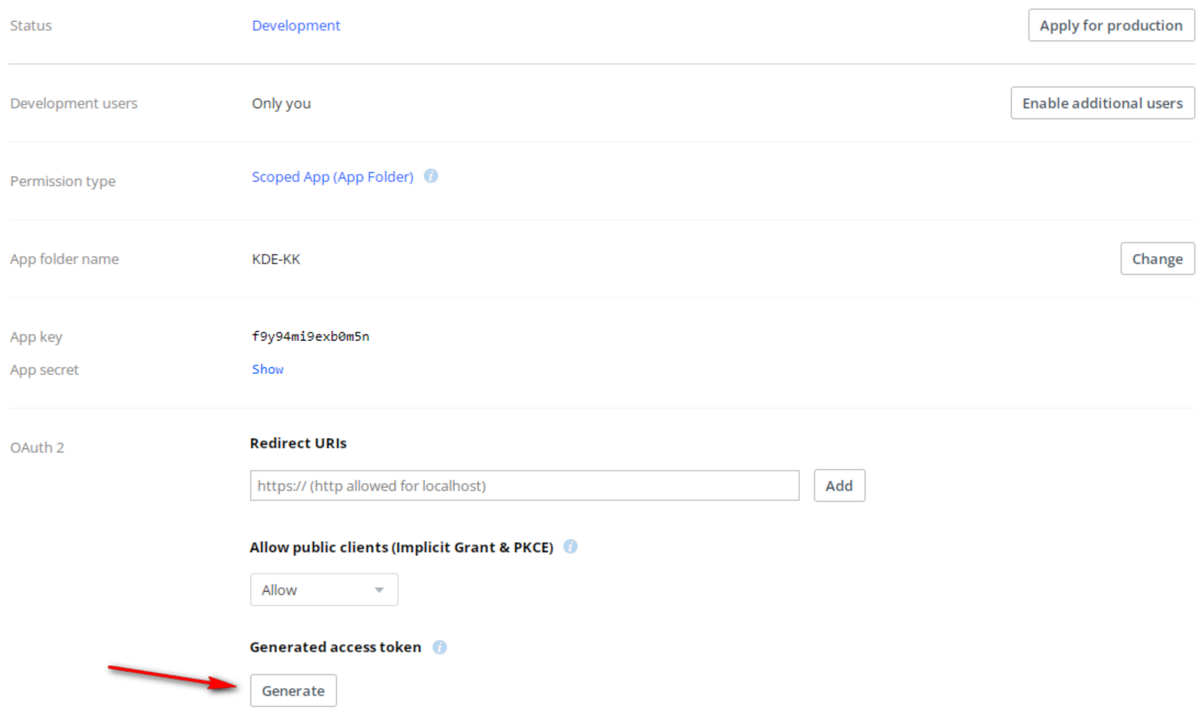
```
dropbox_interval=120
dropbox_filename=data.txt
```

https://www.dropbox.com/oauth2/authorize?client_id=fhdq6ls9utqksem&token_access_type=offline&response_type=code

The example code in KDE_dropbox.c demonstrates how to use the KDE Dropbox API to request and save (to the config.ini file) a Dropbox "refresh-token" by using the auth-code, app-key and app-secret. The "refresh-token" is a non-expiring code that may be used to periodically request a "short-lived access token" which, in-turn may be used to upload files to Dropbox. The example code uses the expiry seconds value provided with the short-lived access token to automatically refresh the short-lived access token when it is close to expiry or if the KDE has been restarted (as the short lived access token is only stored in RAM).

Only if you have an existing, older Dropbox App, that was created prior to the Dropbox's conversion to using short-lived access tokens, then the above steps are not necessary as the access token will not expire. In this case, from the app page, under the Settings tab, press the Generate button to get an OAuth access token for your app. This is a non-expiring token that you can supply to KDE Dropbox API in place of the short-lived access token order to upload data files to your Dropbox App folder.

Note that should you change the app permissions at any time in future you will need to re-generate this OAuth access token.



Status	Development	Apply for production
Development users	Only you	Enable additional users
Permission type	Scoped App (App Folder)	
App folder name	KDE-KK	Change
App key	f9y94mi9exb0m5n	
App secret	Show	
OAuth 2	Redirect URIs https:// (http allowed for localhost) Add Allow public clients (Implicit Grant & PKCE) Allow Generated access token Generate	

KDE Dropbox API functions

The KDE Dropbox API can be found in KDE_dropbox.h.

Note that KDE_https_client_init() must be called to allocate storage for HTTPS session data before any calls to the KDE Dropbox API.

KDE_dropbox_upload_file()

This function can be used to upload a small amount of data to a named file on Dropbox in a single function call.

```
bool KDE_dropbox_upload_file (
        char *access_token  // In  The OAuth access token required to access your
        Dropbox App
        char *filename,     // In  The filename (or path and filename) to create in
        Dropbox
        bool timestamp_name // In  Add the date/time to the filename
        uint8_t *data,      // In  The data to write
        uint32_t length     // In  The length of data to write
        );
```

If timestamp_name is true then the given filename will be modified to include a chronologically alpha-sorting timestamp (of the form YYYYMMDD_HHMMSS) based on the KDE485 real time clock.

For example, if the filename specified is temperature.csv and timestamp_name is true, the resulting file created within your Dropbox app folder could be named temperature20211225_092034.csv.

KDE_dropbox_upload_start ();

This function may be used to start an arbitrary length upload of data file data to your Dropbox account. This allows data to be sent using a start, send_data, send_data, send_data, end sequence.

```
bool KDE_dropbox_upload_start (
        char *access_token,       // In  The OAuth access token required to access
                                         your Dropbox App
        char *filename,           // In  The filename (or path and filename) to
                                         create in Dropbox
        bool timestamp_name,      // In  true to add the date/time to the filename
        uint32_t length,          // In  The total length of data to be written to
                                         the file
        KDE_TLS_HANDLE **handle   // Out The handle to continue writing
        );
```

Note that the handle specified is an output pointer that will receive an allocated TLS session handle for use when continuing the upload with the KDE_dropbox_upload_send_data() and KDE_dropbox_upload_end() functions.

This function returns true if successful and the upload can be continued with calls to KDE_dropbox_upload_send_data() and must be completed with a call to KDE_dropbox_upload_end() to free allocated resources. If it returns false to indicate failure, any allocated resources are released and no further calls are required.

KDE_dropbox_upload_send_data ()

This function can be called any number of times to supply additional data once a file upload has been started using KDE_dropbox_upload_start().

```
bool KDE_dropbox_upload_send_data (
        KDE_TLS_HANDLE *handle,   // In  The handle issued by
                                         KDE_dropbox_upload_start()
        uint8_t *data,            // In  The data to write
        uint32_t length           // In  The length of data to write
        );
```

KDE_dropbox_upload_end ()

This function completes a multi part data upload to a dropbox file that was started by KDE_dropbox_upload_start(). It will receive the Dropbox API response and test for success before freeing any resources associated with the HTTPS session.

```
bool KDE_dropbox_upload_end (
        KDE_TLS_HANDLE *handle,   // In  The handle issued by
                                         KDE_dropbox_upload_start()
        );
```

Dropbox Examples

The example task KDE_dropbox_task.c shows two ways to upload data to dropbox:

1) If a file defined in dropbox_source_filename=name (in config.ini) is found in the KDE filespace, it is uploaded repeatedly to the dropbox path/filename defined in dropbox_filename=name. The example code uses the timestamp_name option to automatically append a timestamp to the name of each file uploaded.

2) If the dropbox_source_filename= parameter is not found, the example code repeatedly uploads a locally defined string dropbox_data. The same timestamp_name option as above is used to create sequentially named files.

Note that dropbox does not know your local filename (the one in the KDE filesystem). The filename is not uploaded with the file data. It takes the filename from the dropbox_filename= parameter in config.ini, with the optionally appended timestamp mentioned above.

The examples use a value DROPBOX_UPLOAD_FILE_CHUNK_SIZE to specify the size of the block sent to dropbox. It was experimentally found that for dropbox to correctly store the file the value needs to be at/below 4k. Values above 1k yield a negligible upload speed improvement.

TCP to Serial Bridge

This function is a data-transparent (protocol independent) TCP to Serial bridge. It creates pairs of KDE485 serial ports and corresponding TCP/IP ports. Up to five pairs can be configured.

KDE485 serial port <-----> TCP/IP port

KDE485 serial ports are numbered 0-4 where ports 1-4 are physical serial ports and port 0 is the USB VCP.

TCP/IP ports are numbered 1-65535. In most systems, many of these are reserved e.g. port 80 is HTTP and 443 is HTTPS. For this application, high port numbers e.g. 10000+ are preferred because e.g. port 5060 is reserved for VOIP.

Up to five connections, corresponding to the five KDE485 serial ports, can be defined and each can independently work in **Client** or **Server** mode. In server mode it will listen for incoming connections to allow clients to connect. In client mode it will connect to a hostname or IP address. Once the connection is established, data can flow freely in both directions between the serial port and network connection.

When a connection is established, the UART buffers are flushed before data transfer is started.

Obviously, any KDE485 serial port configured as described below is not available for use by other KDE485 applications. This includes both user-written code and built-in code e.g. GPS-RTC. The internal "SPI GPS" module is unaffected because it uses a separate serial (SPI) connection.

Configuration - Common for both modes

All configuration uses config.ini. The usual serial port configurations all apply e.g. port1=9600,8,n,1 to set baud rate etc.

The TCP-Serial Bridge function is globally enabled with

```
ethser=1                          ; start the ETHSER RTOS task
```

The mapping between each serial port and its corresponding TCP/IP port is specified as below e.g.

```
ethser_port1=10005                ; connect serial port 1 to TCP/IP port 10005
ethser_port2=10006                ; connect serial port 2 to TCP/IP port 10006
```

Configuration - Client mode

This is configured by specifying the server (IP or hostname) e.g.

```
ethser_server1=a.kksystems.com    ; connect serial port 1 to a.kksystems.com port
                                    10005
ethser_server2=123.124.125.126    ; connect serial port 2 to 123.124.125.126 port
                                    10006
```

A client will attempt to connect to the server (e.g. to a.kksystems.com) continuously, even if its associated serial port has no incoming data. If the connection is dropped, the KDE485 will try to reconnect automatically.

Configuration - Server mode

When a serial port has no IP/hostname specification (no ethser_serverX= entry), the bridge for that port is in Server mode and waits for incoming connections. The serial port still needs to be configured in the Common configuration.

A port configured as server can have only one client connected at any time. When a new client connection comes in, the existing connection is dropped, the serial port buffer is flushed and the new client is connected.

Example 1

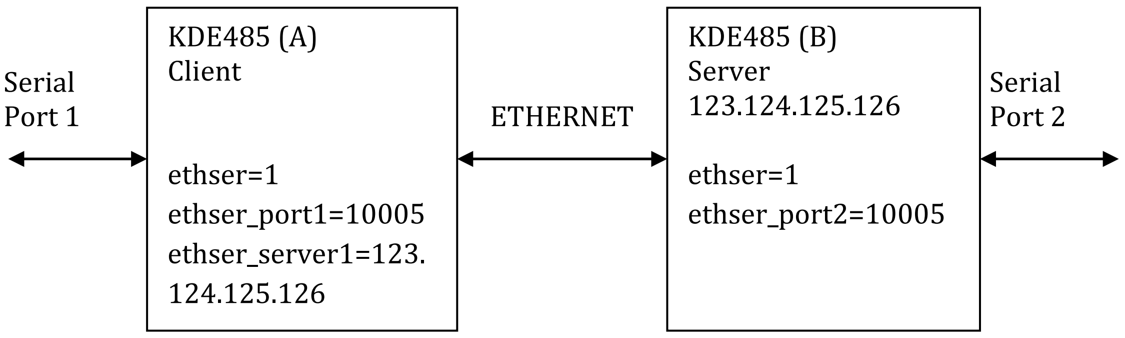
```
ethser_port1=10005
ethser_port2=10006
ethser_server1=a.kksystems.com
```

The above config creates

- a client on serial port 1 connecting to tcp/ip port 10005 on a.kksystems.com
- a server on serial port 2 accepting connections on tcp/ip port 10006

Example 2

This interconnects the serial ports of two KDE485 units, with the data flowing over a LAN or a WAN between them. One KDE485 has to be configured as Client and the other KDE485 as Server. Serial Port 1 of one KDE485 is connected (via ethernet) to Serial Port 2 of the other KDE485.



Note that both client and server must use the same TCP port (10005 in this case).

To discover the IP of the server (123.124.125.126 in the above example) look in boot.txt (if using DHCP) or set a fixed IP in config.ini.

Each of the five KDE485 serial ports can have its Client or Server configuration independently of the others, so you can have up to five connections running concurrently over TCP.

Following power-up, a client will try to establish a TCP connection to the server repeatedly and without a limit on the number of attempts. The link will therefore always "come up" initially even if there is no data arriving at the client's serial port. However, once it has established a TCP connection, and if there is no data flowing, the TCP connection will eventually time out, unless keep-alive is enabled - see the **Keep Alive** section below.

Only a Client can initiate a TCP connection. This system is therefore best suited to Master/Slave applications. The Master is connected to the Client KDE485 and this ensures the TCP connection will always be established when needed.

The "ethernet" connection in the above diagram can be a site-site WAN, VPN, etc. If not using a VPN, observe the usual IOT security procedures regarding not having open ports exposed to the public internet.

The above client/server configurations can be modified at runtime by editing config.ini. Changes are detected every 10 seconds.

Keep Alive

This is configured in config.ini with e.g.

```
ethser_ka=10
```

producing a keep-alive packet every 10 seconds. This setting is read at KDE485 startup only. A value of 0 disables keep-alive. The minimum value is 5.

Keep-alive is a standard TCP/IP feature but its effect varies between systems. Some equipment disregards keep-alive packets. At the most basic level is prevents TCP/IP timeouts when there is no data flowing. It will not re-establish a TCP connection if the server has been rebooted, but new data arriving at the client serial port will re-establish that.

Maximum data rate, Baud Rate considerations, and data integrity

The serial ports have 1k TX and RX buffers, are polled for RX data approximately every 10ms, and any data is transferred to a buffer whose default size is 512 bytes. This buffer size yields a transfer rate of up to 512 / 0.010 = 51200 bytes per second.

The buffer size can be changed in config.ini with e.g.

```
ethser_bufsize=128
```

In reality the data rate will be limited by the serial port baud rates. A further limit is imposed by the lack of flow control on RS232/422/485. The USB VCP (port 0) has flow control but the other factors are likely to dominate.

In Example 2, the serial port 2 baud rate must be at least as high as the serial port 1 baud rate, otherwise data may be lost. Alternatively, a half-duplex protocol can avoid data loss by containing gaps in the data.

For best performance, the TCP/IP Nagle algorithm is disabled in LwIP. However, in some situations, generating tiny ethernet packets at a high rate is undesirable. There are two data aggregation algorithms available and are configurable in config.ini:

```
ethser_agg=0
```

This does not aggregate data and a new ethernet packet is sent every 10ms, if there is data to send. The size of the buffer (default size 512) does not have much of an effect unless you are running at very high baud rates.

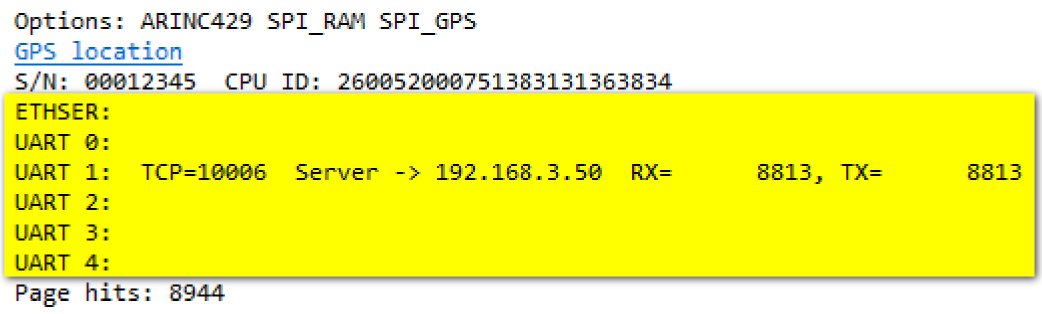
ethser_agg=1

This is the default mode. It works much better on slow networks, WANs, VPNs, etc. It generates a new ethernet packet only when the serial input data has stopped for 10ms, or the buffer (default size 512) has filled up. This mode produces fewer ethernet packets but there can be a greater latency, which can be minimised by keeping the buffer small e.g. 64 bytes with a 9600 baud input produces a latency of around 64ms. There are also cases of ethernet switches with QOS features which interfere with small high frequency packets.

While TCP/IP implements error correction and flow control, this does not always work on WAN links. 3G/4G links can have intermittent latencies in seconds. Serial comms does not have error correction and errors are therefore possible. An upper level protocol is therefore desirable if you want guaranteed data integrity, and in half duplex applications generous timeouts need to be configured in the system doing the polling. Short packets (a few hundred bytes, or less) help because they are more likely to fit inside one MTU, especially if multiple concurrent channels are being run.

Status Monitoring

If ethser=1 in config.ini, the HTTP Server **Status** page shows the current status of the TCP-serial bridge function:



```
Options: ARINC429 SPI_RAM SPI_GPS
GPS location
S/N: 00012345  CPU ID: 2600520007513831313638 34
ETHSER:
UART 0:
UART 1:  TCP=10006  Server -> 192.168.3.50  RX=      8813, TX=      8813
UART 2:
UART 3:
UART 4:
Page hits: 8944
```

The above shows that there is a Server on serial port 1, connected to a Client on 192.168.3.50, and 8813 bytes have been transferred in both directions.

Applications for the TCP-Serial bridge include:

- Remote polling of half duplex instruments over a LAN or WAN
- Making a device with a serial port accessible over a LAN or WAN
- Extending a serial port over a long distance using two KDE485 modules
- Allowing a device with a serial port to access a network device

Software Versions

LWIP: V2.0.3
MbedTLS: V2.16.2
FreeRTOS: V10.0.1
FatFS: V0.12c

The KDE485 is designed and manufactured in the UK.

KK Systems Ltd
Tates
London Road
Pyecombe
Brighton
BN45 7ED
Great Britain

tel +44 1273 857185
kksystems.com
sales@kksystems.com