weAut_01 / weAutSys

R 2.2.1

Generated by Doxygen 1.8.0

Wed Jan 25 2017 17:36:20

Contents

Chapter 1

weAutSys - automation controller system software

Please see below notes on Copyright and for non English readers.

1.1 weAut_01 — Compact automation module

weAut_01 is an appliance for digital control and automation. Being the main target hardware for weAutSys it is a very compact (single board) micro-controller based module.

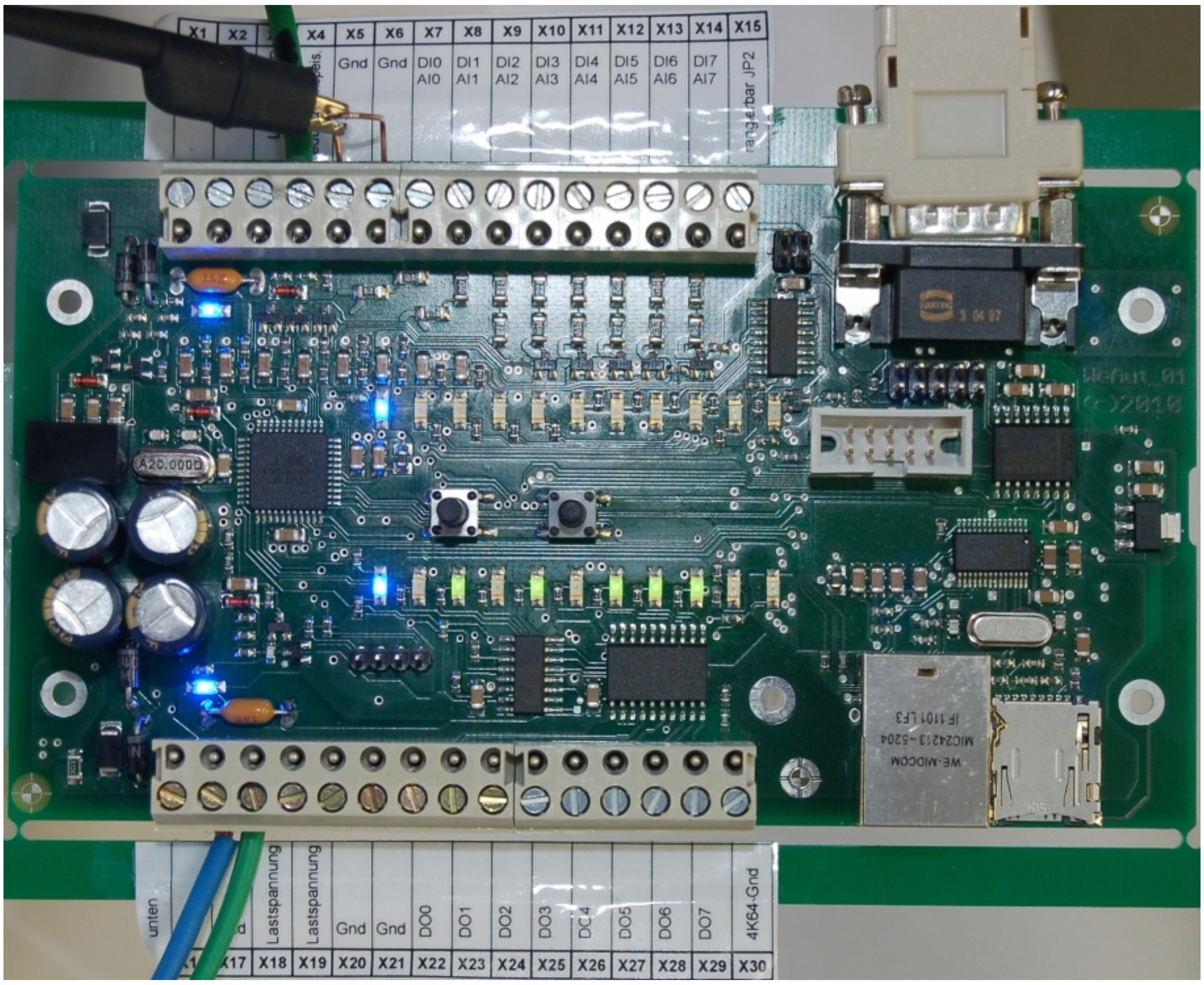

Figure 1.1: weAut_01 automation controller

weAut_01's features are

- compact module sold as board, well fitting in a wide range of compact cases, like e.g. a DIN rail (TS 35) / top-hat rail box
- industry standard voltages (12V, 24V) for supply and process IO
- wide supply voltage range for the process IO (load voltage, or **LV** for short) :
  9..30 V (LV) cover the range of
  - 12V : home / facility automation, automotive (small, passenger) and
  - 24V : industrial PLC, field I/O, automotive (lorries, trucks)
- low loss supply of the micro-controller and other electronic (5V, 3.3V) circuitry from the LV
- optional second redundant 9 .. 30 V feed for the electronics, suitable for uninterrupted battery supply or even AC (12..18V∼)
- fit for "green automation" thanks to low power consumption and efficient run time
- 8 digital output (DO) channels (LV, 100mA, protected)
- 8 protected input channels for +/-70V input voltage (surviving 250 Veff as absolute max. rating)
  Every channel can be configured as
  - digital input
    - in three threshold / hysteresis modes for different (12V / 24V) sensors
  - analogue input
    - in three different single ended voltage ranges
- standard communication links, as
  - V.24 all usual modes, 2400 .. 500Kbaud, optional flow control
  - Ethernet
- extensibility by SPI, I²C (twoWire) etc.
- widely-used micro-controller / architecture with a broad base of tried and tested software and tools
- on board programming via ISP
- persistent storage by embedded EEPROM and a slot for small memory cards

Being compact and low power (green automation) weAut_01 nevertheless has some other typical PLC features:

- protected process IO with error detection
- supervision of its supply (LV)
- runtime with usual automation "cycles" (1ms, 10ms, 100ms, 1s) known from most PLCs

1.2 weAutSys — Automation runtime system

weAutSys is the runtime / operating system for weAut_01 as well as for comparable hardware.

Ports to other µControllers and boards (ArduinoMega2560, ArduinoUno) have been done and others are on their way. According to their processing power and features the range of functions of those evaluation boards is smaller compared to weAut_01.

This weAutSys runtime gives the application / automation programmer real time features, threading, the usual automation "cycles" (1ms, 10ms, 100ms, 1s) known from most PLCs as well as timers, a full grown TCP/IP stack, small memory card, serial communication and more.

Hint: The clock driven cyclic execution (in automation language) are periodic tasks (in IT / real time language). To to most of the application work in those "cycles" is an approach symptomatic for PLCs.

As a subordinate via network connection to a central processing computer weAut_01 is a good choice for process signal concentration tasks plus optional pre-processing and local automation. weAut_01 is also suitable for small automation and control applications, networked and stand alone.

In that field of application I/O signals conforming to industry standards (i.e. signal and protection levels) are as vital as the ability to communicate via standard media and prevalent open protocols (Ethernet, TCP/IP, DHCP, NTP, Telnet, Modbus etc.).

To sum up

- weAutSys supports all of weAut_01's features
- existing professional software / architecture projects — like uIP, Protothreads and fatFS having free non-infecting licenses — were ported to / modified for the commercial project weAutSys.
- implementing the (embedded) controller software solely in C
- using a "SVN — Eclipse — AVR-GCC" tool chain
- on board programming via ISP
- usual automation "cycles" — 1ms, 10ms, 100ms, 1s — known from most PLCs / automation systems
- handling of date, time, zones and daylight saving, optionally (preferably) by information got as DHCP and/or NTP client
- absolute (date time) and duration timers
- handling of process inputs analogue, digital, counter etc.
- handling and supervising of digital process outputs
- handing of communication and command line interfaces (CLI)
- handling and supporting a wide range of protocols: DHCP, NTP, Telnet, Modbus and others
- supervising of "load voltage" (LV) supply
- multi-treading
- watchdog supervision of application / user software
- logging of (last) re-start cause
- a serial bootloader and its (optional) integration with the application program

1.3 Threads and "cycles"

All user software is organised in application threads and most system software runs in system threads.

Those application and system threads are implemented as protothreads

Predefined application threads to be optionally filled with life by the user / application programmer are

- cyclic threads — 1ms, 10ms, 100ms, 1s — for tasks to run regularly at that intervals (or its multiples).

  These predefined periodic tasks implement the usual clock driven "cyclic" execution of PLCs.
- event threads like
  - the CLI (command line interpreter) thread where the application programmer can handle (or ignore) textual messages to the system,

  - a system startup thread
    and others

- helper threads, that can be started as child of any other thread to handle sub-tasks, -protocols, -state machines and the like.

  Some of those helpers are pre-defined, like e.g.

  - output of system version info,
  - output of system runtime / state info
    and others

  More helper threads can freely and easily be added by the application programmer.

1.4 Process I/O

weAut_01 has

- 8 digital output (DO) channels (LV, 100mA, protected) and
- 8 digital protected input (DI) channels.
- analogue inputs (AI). Each of the 8 DIs may be used as a simple (i.e. single ended and unipolar) analogue input also.

Besides that some extra extensions — counter, generator etc. — are available.

The I/O channels' and drivers' state is or can be displayed by an appropriate number (about 20) of status LEDs.

weAutSys' process and HMI I/O support function make the utilisation of those features an easy task. See Process I/O (support) for details.

1.5 Serial communications

The serial link (UART, RS232) for a normal terminal connection that is usually text-based as HMI (mam-machine-interface). Then as default the serial link will be tight to the C standard streams stdin, stdout and stderr. So these streams plus their standard formatters resp. parsers could be used for external logging and command line interface at standard serial terminals or terminal emulating programs (like HTerm). This can prove very useful for commissioning and maintenance.

See modules Serial communication and Serial I/O for technical details.

The signals CTS and RTS are available via two ports and can optionally be used for flow control. Alternatively these signal connections (SubD 9) can be used as counter input respectively generator (PWM) output or just as extra I/O if by cabling separated from the serial RX and TX.

Of course, RX and TX can also be used as extra as extra process input and output too, then sacrifying the RS232 communication. And, of course, the serial RS232 link can serve (non human, non text based) process control applications like e.g. S5/S7 connection via aRS3232/RS485 converter or HART sensors/actors via a suitable HART modem.

1.6 Ethernet communications

The Ethernet / LAN communication is based on

- a ENC28J60 interface (hardware),
- an event driven 28J60 driver and

- uIP [by Adam Dunkels] as TCP/IP stack.

At the higher application (support) layers weAutSys supports and uses

- the address resolution protocol ARP,
- the dynamic host configuration protocol DHCP,
- the Network time protocol NTP,
- a Telnet server and
- a Modbus server.

The DHCP client asks for up to two NTP servers which can then be used by the NTP client function. This (best) practice will synchronize the time of weAut_01 modules in one network within 20 ms or — usually — much better.

The TCP/IP stack and drivers support, besides said ARP, DHCP, Telnet, Modbus and NTP, the implementation of higher level protocols and applications, some of them already brought along with uIP.

1.7 Small memory card

A card slot allows a wide range of small memory cards (SMCs) to be used for application specific purposes, as e.g. logging for stand alone systems or in case of communication outages.

The system software supports the basic access to small memory cards via SPI and higher level functions support the necessary subset of FAT file systems if one is found on the small memory card inserted. The natural choice and often standard for SMCs is FAT32.

Hint: Manufacturers of microSD cards accessible via SPI will have paid the required per card license fee do to so. For license reasons be sure to embed only those SMCs in your weAutSys based based automation modules like weAut_01 using a supported (FAT) file system (with short filenames).

For persistent storage of configuration data the use of the embedded EEPROM is the better choice.

1.8 A note on Copyright and License

For hardware, software and documentation developed for this project:

weAutSys is commercial software. Licenses to use are sold with the products that embed it and are bound to it.

Other license variants are available.

Some parts of the system are modified open source software with non-infecting BSD (like) license.

For a source (file) of that kind — and only for those — the following holds:

It keeps its original license and is hence still open source.

Those modified source files are available as download. See the comments and license notes there.

1.9 A note to non anglophone readers

All weAutSys' developer / software documentation is in English. The only exception is the following Overview in German.

Die Entwicklungsdokumentation von weAut_01 und weAutSys ist in Englisch, bis auf den kurzen deutschen Überblick (Overview).

1.10 Überblick (Overview in German language)

weAut_01 ist eine Mikrocontrollerbaugruppe mit industrietauglichen Prozess-I/O-Schnittstellen und Standard--Kommunikation über Ethernet und V.24.

Die Baugruppe eignet sich so direkt für Automatisierung — auch "stand alone", sowie vernetzt auch für eine vorverarbeitende Prozesssignalerfassung und -verteilung, die in einem Netzwerk zentralen Rechnern unterlagert ist. Hierfür ist die industriegerechte I/O genauso entscheidend wie die Kommunikation über verbreitete, auch preiswerte, Medien und übliche Protokolle: Ethernet, TCP/IP, DHCP, DNS, NTP, Telnet, Modbus etc..

Einige Eigenschaften

- verbreiteter Mikrocontroller mit breiter Basis an bewährter Software und Werkzeugen
- bewährte Software (wie uIP, Protothreads, fatFS)

  Diese werden dank mit freier Lizenz auch in diesem kommerziellen Projekt — i.A. tiefgreifend modifiziert — genutzt.
- Erstellung der (embedded) Controller-Software ausschließlich in C
- SVN, Eclipse, AVR-GCC tool chain als professionelle und lizenzgebührenfreie Werkzeuge
- on board Programmierung auch ohne zusätzliche Geräte
- kompakte Baugruppe mit Einbaumöglichkeiten in Verteiler- und (DIN-) Hutschienengehäusen
- großer Spannungsversorgungsbereich für die Versorgung der Prozess-I/O, also für die sog. Lastspannung (UL)
- 9..30 V (UL) überdecken 12V- und 24V-Anwendungen und -Peripherie
- verlustarme Versorgung der µController und der übrigen Elektronik aus der UL und / oder einer zusätzlichen redundanten Einspeisung

Hardwarebasis ist ein ATmega1284P als Mikrocontroller und ein ENC28J60 für die MAC- und PHY- Schicht des Ethernet.

Genauso wie für den zugehörigen seriellen bootloader existieren das weAutSys runtime System Portierungen auf andere µController und Module, wie z.B. für ArduinoMega2560, ArduinoMegaADK und ArduinoUno. Teile davon, wie u.a. der Bootloader, wurden im Sommer 2013 als open source verfügbar gemacht.

Für die im Rahmen des Projekts entwickelte Hardware, Software und Dokumentation gilt:

Chapter 2

Module Documentation

2.1 + + + weAutSys -- the runtime for automation µ-controllers + + +

2.1.1 Overview

weAutSys is a runtime for small network enabled and µ-controller based automation modules like weAut_01. weAutSys's version in hand is for Atmel ATmega µ-controllers, that is a Harvard RISC machine.

See the introduction for more details.

As **RAM** is a precious resource in most µ-controllers it is recommended to confine constant structures, invariant strings and arrays etc. to program or "flash" memory. But alas, albeit designed for hardware / system / core related programming (and in the proximity / as a better substitute for assembler) the C programming language was never made fit to handle memory architecture features like segmentation (Intel) or Harvard (ATmega).

And most C implementations, like the GNU C Compilers and tools (GCC) used here, do not mend these deficiencies. To handle ATmega's Harvard architecture one has to use the ugly helper constructs (<avr/pgmspace.h>) instead of just writing plain C assignments and references. Having to handle ATmega's Harvard architecture with add-ons has side effects to other tools, certain Doxygen problems being one / the least of them.

The second scare resource with low electric power consumption µ-controllers is **processing time**. Therefore more people than one may believe tend to use Assembler more than C. This might have made sometimes sense in the past. The reasons to make weAutSys with C are:

a) Clever algorithms are the easier to implement the higher the language level.

b) Compiler technology is quite advanced now. GCC for AVR does clever instruction and register usage — sometimes to an astonishing extend and anyway better than a human could provide in Assembler over a whole project.

But b) is, most regrettably, not true in all (algorithmic / syntactic) circumstances. If reading the generated code uncovers a total flop not remediable by changing C constructs the "in line Assembler" is used locally and sparingly. GCC's inline ASM is obstinate and no fun at all to use. On the other hand it fits in or does not break C Compiler optimisations.

So the weAutSys runtime is an almost pure C project.

Modules

- + + System services (threading, time keeping, communication) + +
- + + Application (layer) support + +
- + + Utilities and helpers + +
- + + Process I/O and HMI functions + +
- + + Low level system services + +
- + + Bootloader support + +

Defines

- #define EEP_SIZE (E2END + 1)

  The EEPROM size in byte.
- #define USE_BOOTLOADER USE_BOOTLOADER_PRESET

  Have and use a bootloader.

2.1.2 Define Documentation

2.1.2.1 #define USE_BOOTLOADER USE_BOOTLOADER_PRESET

Have and use a bootloader.

This value determines the existence and usage of a bootloader:

0: no bootloader or at least no bootloader integration 1: bootloader area used / bootloader existing 2: bootloader fully integrated: application programme may use all No Read-While-Write (NRWW) flash section functions and constants.

Fully integrated means that all µController resets go through the bootloader and the bootloader will not enter the program (=x00000) without having done all basic initialisations. The latter hence can be omitted from system/application software. Full integration also means that the bootloader may be entered by (CLI) command.

2.2 + + System services (threading, time keeping, communication) + +

2.2.1 Overview

weAutSys provides a bundle of system function and variables for use by the application software (organised in threads).

Great care should be taken by the application programmer to never violate the the usage conditions stated in the documentation; like not to modify variables described as read only for the user software, obeying the threading rules and so on.

Modules

- User / application threads
- System threads
- Timer handling
- Time keeping (date and time)
- Serial communication
- LAN / Ethernet communication
- Communication with a small memory card (via SPI)

Files

- file system.h

  weAutSys definition of system calls and services to be used by application / user software

2.3 User / application threads

2.3.1 Overview

In weAutSys user respectively application specific software comes as threads.

These threads as well as the system threads are implemented as protothreads. The following user / application threads are or may be defined

1. an application initialisation thread
2. an 1 ms application thread
3. a 10 ms application thread
4. a 100 ms application thread
5. an 1s application thread (see registerAppThread() for all four cyclic threads)
6. an application serial input processing thread (see registerAppSerInpThread()).
   In case of HMI text I/O it would delegate the work a supported / supplied CLI (command line interpreter).
7. an application background tasks thread

A weAutSys / weAut_01 application is implemented by writing one or more of those threads (and usually nothing else). The weAutSys' user, customer or application programmer provides these threads by writing a protothread function of type `ptfnct_t f()` (see pt.h).

From list above the first one, the initialisation thread function appInitThreadF(), must be thus provided, even if doing almost nothing.

And at least one of the others has to be implemented and registered in the appInitThreadF() function. Otherwise the whole thing would run without any process control effects to the external world interfaces.

2.3.2 Rules for user threads' CPU usage

No thread must run in one flow longer than 20 % of its tick period or 3 ms, whatever is shorter.

The said 3 ms limit is critical and must be reduced to 300 µs ∗2) if the following should hold:

- The application software relies significantly on its 1 ms cycle respectively thread.
- This 1ms thread is not / cannot be made able to handle several (2..5) past ticks in one step if scheduled with a flag saying so.

If a task takes longer as the time limits stated above, the thread must interrupt itself by Protothreads blocking or yielding operations ∗3) at due intervals. A software running longer than 100 ms without yielding will effectively detriment the work of the system. Running longer than 250 ms without yielding will cause a reset, i.e. abort.

The background of these basic rules is that Protothreads and hence weAutSys is a non preemptive OS (for 1 CPU core). So no (user or system) thread will take over from another except at the said Protothreads operations, i.e. yielding, blocking and exiting. Nor will any thread run (really or quasi) parallel with another. This simplifies a big lot ∗1), at the cost of the strict rules for processor time usage above.

Remark 1: The only remaining complication, the interrupts, are handled by weAutSys. It is highly discouraged for customer software to introduce additional interrupts. It should be kept in mind that Protothreads operations and semaphores are "intra-Protothreads" only and not suitable to synchronise with interrupts.

Remark 2: If the 1ms application thread is critical in said (300 µs) sense, refrain from using SMC and especially file system operations while automation cycles are on.

Remark 3: This set of rules put a lot of trust and responsibility to the application programmer. This is adequate for embedded automation systems in the light of the software quality required for process control systems anyhow. This assumption would fail in a universal purpose computer system (like a PC).

Functions

- ptfnct_t app100msThreadF (struct mThr_data_t ∗uthr_data)

  The user / application specific 100 ms thread.
- ptfnct_t app10msThreadF (struct mThr_data_t ∗uthr_data)

  The user / application specific 10 ms thread.
- ptfnct_t app1msThreadF (struct mThr_data_t ∗uthr_data)

  The user / application specific 1 ms thread.
- ptfnct_t app1sThreadF (struct mThr_data_t ∗uthr_data)

  The user / application specific one second thread.
- ptfnct_t appBgTskThreadF (void)

  The user / application specific background task thread (the function)
- ptfnct_t appInitThreadF (struct pt ∗pt)

  The user / application specific initialisation thread.
- ptfnct_t appSerInpThreadF (struct thr_data_t ∗serInpThread)

  The user / application serial input processing thread.
- p2ptFun registerAppBgTskThread (p2ptFun threadF)

  Register a user / application background task thread.
- p2ptFunC registerAppSerInpThread (p2ptFunC threadF)

  Register a user / application serial input processing thread.

Variables

- struct mThr_data_t app100msThread

  The user / application 100ms thread.
- struct mThr_data_t app10msThread

  The user / application 10ms thread.
- struct mThr_data_t app1msThread

  The user / application 1ms thread.
- struct mThr_data_t app1sThread

  The user / application 100ms thread.
- struct mThr_data_t appBgTskThread

  The user / application specific background task thread.
- struct thr_data_t appSerInpThread

  The user / application serial input handling thread.

2.3.3 Function Documentation

2.3.3.1 p2ptFunC registerAppSerInpThread (p2ptFunC *threadF*)

Register a user / application serial input processing thread.

The thread so registered will be scheduled if the serial input buffer contains a line feed (and hence a line to interpret as command) or is nearly full.

It is expected that the (user's / customer's) implementation reads one line (resp. all the input causing the near overflow) and interprets it according to its own specification. The function setCliLine() can be used to hand an input line to system and application command line interpreters.

Parameters

threadF	the thread function to register;

Returns

the previously registered thread function

See also

appSerInpThreadF
appInitThreadF
uartGetLine(char line[], int max)

Examples:

main.c.

2.3.3.2 ptfnct_t appSerInpThreadF (struct thr_data_t ∗ *serInpThread*)

The user / application serial input processing thread.

If something is to be done in the user / application software for characters or lines received via serial input an input processing thread (i.e. a protothread function) must be provided therefore in an application specific source file and that thread function has to be registered according to this declaration.

For that function any name can be chosen. This declaration is provided just for convenience and harmonisation.

The parameter this function (if registered) would be called with will (at present) point to appSerInpThread. Its flag will be set by system software as described there. Its up to the implementation to read (buffered) UART input and act on it accordingly. If the UART connects to a human operated terminal one would normally use the the variable part of the parameter thread structure as command line interpreter (CLI) structure by help of the setCliLine() function.

Parameters

serInpThread	

See also

appSerInpThread
registerAppSerInpThread(p2ptFun)
appInitThreadF

2.3.3.3 p2ptFun registerAppBgTskThread (p2ptFun *threadF*)

Register a user / application background task thread.

The thread so registered thread will be scheduled at lowest priority.

Parameters

threadF	the thread to register;

Returns

the previously registered thread function

See also

appBgTskThreadF
appInitThreadF

2.3.3.4 ptfnct_t appBgTskThreadF (void)

The user / application specific background task thread (the function)

If something can or has to be done with the lowest priority (and in CPU usage partitioning $<$ 2ms (!)) it may be done the background thread.

In that case a protothread function must be provided therefore in an application specific source file and that thread function has to be registered accordingly.

For that function any name can be chosen. This declaration is provided just for convenience and harmonisation.

Note

It is really essential that the background thread function will yield, block or PT_EXIT exit after less than 2 ms (better $<$ 1ms) CPU usage. If no work is to be done it should do
return PT_EXITED;
even before the
PT_BEGIN(&appBgTskThread.pt);
Note also that the background thread is also scheduled in the state PLCstop. If automation tasks are done in this thread the PLC state should be checked by PLCinRUN (or PLCinSTOP).

See also

registerAppBgTskThread(p2ptFun)
appInitThreadF

2.3.3.5 ptfnct_t appInitThreadF (struct pt ∗ *pt*)

The user / application specific initialisation thread.

This protothread function must be provided in an user / application specific source file. This thread is scheduled in one to three phases divided by yield points or until it exits.

Phase 1: pre system init.

In this phase user variables and data structures may be set as well as system variables controlling system initialisation.

Phase 2: post system init.

In this phase serial communication is available.

Phase 3: post persistence init.

In this phase persistent configuration data have been fetched and may be modified before being used in following phases.

Phase 4: post Ethernet init.

In this phase the driver and the stack are initialised. This does not necessarily mean the link is up or the valid IP address is set already.

Nevertheless communication might be prepared and listen calls might be made here.

Phase 5: post ? ? (future use) init.

In this phase is presently empty.

Phase 6: reschedule until exited.

From this phase on the initialisation thread will be rescheduled until it exits. This allows for any initialisation work whatever complicated or time consuming.

Once exited the initialisation thread will never be rescheduled. This is already true for all above phases.

It is essential that all user threads (cyclic, CLI etc.) be registered here.

2.3.4 Settings of variants

The weAut_01 board allows some hardware variants by

- choosing other components or even just omitting some
- setting of jumpers
- connecting extensions to jumpers / connectors
- usage or non-usage of certain micro-controller features, like
  AD, timer, counter, generator, 2wire, UART, flow control,
  on the respective I/O.

The user's initialisation thread is the (main) place to adapt to such and other variants.

Parameters

pt	pointer to protothread structure

2.3.4.1 ptfnct_t app1sThreadF (struct mThr_data_t ∗ *uthr_data*)

The user / application specific one second thread.

If something is to be done in the user / application one second thread a protothread function must be provided therefore in an application specific source file and that thread function has to be registered accordingly.

For that function any name can be chosen. This declaration is provided just for convenience and harmonisation.

See also

registerAppThread(struct mThr_data_t ∗, p2ptFun)
app1sThread
appInitThreadF

Parameters

uthr_data	pointer to extended protothread structure

setStatusLedRd(ON); // Test: only blink blink

setTestPin1(OFF); // Test: execution time measurement

2.3.4.2 ptfnct_t app100msThreadF (struct mThr_data_t ∗ *uthr_data*)

The user / application specific 100 ms thread.

If something is to be done in the user / application 100 milliseconds thread a protothread function must be provided therefore in an application specific source file and that thread function has to be registered accordingly.

For that function any name can be chosen. This declaration is provided just for convenience and harmonisation.

See also

registerAppThread(struct mThr_data_t ∗, p2ptFun)
app100msThread
appInitThreadF

Parameters

uthr_data	pointer to extended protothread structure

2.3.4.3 ptfnct_t app10msThreadF (struct mThr_data_t ∗ *uthr_data*)

The user / application specific 10 ms thread.

If something is to be done in the user / application 10 milliseconds thread a protothread function must be provided therefore in an application specific source file and that thread function has to be registered accordingly.

For that function any name can be chosen. This declaration is provided just for convenience and harmonisation.

See also

registerAppThread(struct mThr_data_t ∗, p2ptFun)
app10msThread
appInitThreadF

Parameters

uthr_data	pointer to extended protothread structure

(! doDriverOK())

2.3.4.4 ptfnct_t app1msThreadF (struct mThr_data_t ∗ *uthr_data*)

The user / application specific 1 ms thread.

If something is to be done in the user / application 1 milliseconds thread a protothread function must be provided therefore in an application specific source file and that thread function has to be registered accordingly. Best care should be taken to have the CPU load between two yield or block points well below 150µs.

For that function any name can be chosen. This declaration is provided just for convenience and harmonisation.

See also

registerAppThread(struct mThr_data_t ∗, p2ptFun)
app10msThread
appInitThreadF

Parameters

uthr_data	pointer to extended protothread structure

2.3.5 Variable Documentation

2.3.5.1 struct mThr_data_t app10msThread

The user / application 10ms thread.

The flag value is the number of periods the scheduled thread should work on. It is normally 1 except for delays due to processor overload. The flag must be reset by the thread function

Examples:

main.c.

2.3.5.2 struct mThr_data_t app1msThread

The user / application 1ms thread.

The flag value is the number of periods the scheduled thread should work on. It might well be greater than 1 as processor load might very well inhibit this thread to be scheduled at every single milliseconds tick. The flag must be reset by the thread function.

2.3.5.3 struct mThr_data_t app100msThread

The user / application 100ms thread.

The flag value is the number of periods the scheduled thread should work on. It is normally 1 except for delays due to processor overload. This flag must be reset by the thread function.

Examples:

main.c.

2.3.5.4 struct mThr_data_t app1sThread

The user / application 100ms thread.

The flag value is the number of periods the scheduled thread should work on. It is normally 1 except for delays due to processor overload. This flag must be reset by the thread function.

Examples:

main.c.

2.3.5.5 struct thr_data_t appSerInpThread

The user / application serial input handling thread.

This thread handles serial (UART0) input if the user software registers an appropriate function therefore.

The flag will be set by system software, whenever

a) the UART receives a line feed or return code, i.e. when uartInRetBufferd() would return >= 1 or

b) the UART input buffer is filled to (near full) a level, i.e. when the the sender would be stopped by flow control (if activated).

appSerInpThread.flag will be cleared by system software if the UART input buffer gets empty.

It is up to the registered user thread function to reset the flag on other conditions if appropriate.

Examples:

main.c.

2.4 System threads

2.4.1 Overview

System threads as well as the user / application threads are implemented as protothreads. A system thread runs regularly or event driven and handles basic operation system (OS) tasks and services.

Normally to the user / application software (writer) the system threads are of no concern. As a rule a periodic ("cyclic") system thread is always scheduled before a user thread of the same frequency and it is that system thread that sets or increments the user thread's flag to signal the periodic task is due. So before this due signal all system time keeping and else duties can be assumed done.

Files

- file syst_threads.h

  weAutSys' system threads

Defines

- #define ABSsecADJ 0x80

  absolute date time seconds were adjusted
- #define ABSsecSET 0x40

  absolute date time seconds were set
- #define exitNotFlaggedThread(thread)

  Exit a not flagged user / application thread.
- #define NO_SPMIN_SAMPLE 0

  Do not sample the stack pointer and update its minimal value.
- #define resetAppThread(thread)

  Reset respectively initialise an user / application thread.
- #define runOutSysInfoThread(parent, pt, toStream)

  Start (spawn) the outSysInfoThreadF thread as child.
- #define SAMPLE_MIN_SP(nonc)

  Sample the stack pointer and update its minimal value.
- #define scheduleAppThread(thread)

  Unconditionally schedule an user / application thread.
- #define scheduleFlgdAppThread(thread)

  Schedule a flagged user / application thread.
- #define scheduleRegdAppThread(thread)

  Schedule a registered user / application thread.
- #define scheduleYldAppThread(thread)

  Schedule a flagged or yielding user / application thread.
- #define SINCsINCR 0x01

  run since seconds were incremented

Functions

- uint8_t addr3Here (void)

  Get the upper (bit 16 ...) part of the current address.
- uint32_t addrHere (void)

  Get the current address.
- void goto_P (void ∗labelAsValue)

Go to a label (as value) and stay within the 64Kword page.
- void initOutFlashTextThread (outFlashTextThr_data_t ∗outFlashTextThread, char const ∗const ∗theFlash-Strings2out, uint8_t noOfFlashStrings2out)

  Initialise the output a flash string array to standard output thread.
- void initSetAppThread (struct mThr_data_t ∗thread, p2ptFun threadF)

  Register and (force) init an user / application thread.
- void initThreads (void)

  Initialise the threads.
- ptfnct_t outFlashTextThreadF (outFlashTextThr_data_t ∗outFlashTextThread, FILE ∗toStream)

  Output a flash string array to standard output thread.
- ptfnct_t outSysInfoThreadF (pt_t ∗pt, FILE ∗toStream)

  Output some system (version) info, the thread function.
- p2ptFun registerAppThread (struct mThr_data_t ∗thread, p2ptFun threadF)

  Register an user / application thread.
- ptfnct_t sys100msThread (void)

  The 100 milliseconds system thread, the function.
- void sys1msThread (void)

  The milliseconds system function (or thread)
- ptfnct_t sys1sThread (void)

  The 1 second system thread, the function.

Variables

- uint16_t minStckP

  Minimal stack pointer value sampled.
- struct pt ptSys100msThread

  The 100 milliseconds system thread, the (raw) Protothreads datastructure.
- struct pt ptSys1sThread

  The 1 second system thread, the (raw) Protothreads datastructure.
- uint8_t sys100msPeriodFlag

  Flag for system 100ms thread.
- uint8_t sys1sPeriodFlag

  Flag for system 1s thread.

2.4.2 Define Documentation

2.4.2.1 #define **NO_SPMIN_SAMPLE 0**

Do not sample the stack pointer and update its minimal value.

To hinder the sampling the minimal stackpointer define the macro NO_SPMIN_SAMPLE with a value of 1.

2.4.2.2 #define **SAMPLE_MIN_SP(** *nonc* **)**

Sample the stack pointer and update its minimal value.

The minimal stackpointer will be updated at the point of calling this, if the macro NO_SPMIN_SAMPLE is undefined or 0.

Parameters

nonc	number of nested function calls within embedding function The value should be in the range 0..3. If the call hierarchy goes deeper or the functions called use local variables those functions called should sample self.

2.4.2.3 #define runOutSysInfoThread(*parent,* pt, *toStream*)

Start (spawn) the outSysInfoThreadF thread as child.

Parameters

parent	(type pt) the calling parent's thread raw (!) stucture
pt	the thread's to be called raw (!) stucture
toStream	streams to switch the standard streams to (if not MULL)

Examples:

main.c.

2.4.2.4 #define resetAppThread(*thread*)

Reset respectively initialise an user / application thread.

This is a macro: Do set the parameter as name (i.e. `the_thread`) and not as a pointer to resp. address of (not `&the_thread`).

Parameters

thread	(type mThr_data_t) the thread to reset.

2.4.2.5 #define scheduleRegdAppThread(*thread*)

Schedule a registered user / application thread.

This macro will schedule the thread provided as parameter if it has a thread-function registered.

Besides checking this condition its like scheduleAppThread(thread).

Parameters

thread	(type mThr_data_t) the thread to schedule; must not be NULL.

2.4.2.6 #define scheduleAppThread(*thread*)

Unconditionally schedule an user / application thread.

This macro will schedule the thread provided as parameter. As this is done unconditionally the caller is responsible for two hard pre-conditions

1. thread must not be NULL
2. thread.regd must be true

This is a macro, set the parameter as name (thread) and not as pointer to / address of. (That means not use &thread)

Parameters

thread	the thread to reset; must not be NULL.

2.4.2.7 #define scheduleFlgdAppThread(*thread*)

Schedule a flagged user / application thread.

This macro will schedule the thread provided as parameter if its flag is set (and it has a thread-function registered).

Besides checking this condition its like scheduleRegdAppThread(thread).

Parameters

thread	the thread to schedule; must not be NULL.

2.4.2.8 #define scheduleYldAppThread(*thread*)

Schedule a flagged or yielding user / application thread.

This macro will schedule the thread provided as parameter if its flag is set or it has ended by yielding or waiting on its last schedule.

Besides checking this ORed run conditions its like scheduleFlgdAppThread(thread).

Parameters

thread	(type mThr_data_t) the thread to schedule; must not be NULL.

2.4.2.9 #define exitNotFlaggedThread(*thread*)

Exit a not flagged user / application thread.

This macro will exit the thread provided as parameter if its flag is not set.

See also

PT_EXIT
PT_EXITED

Parameters

thread	(state type mThr_data_t) the thread to conditionally exit; must not be NULL.

Examples:

main.c.

2.4.3 Function Documentation

2.4.3.1 void goto_P (void ∗ *labelAsValue*)

Go to a label (as value) and stay within the 64Kword page.

This procedure is a helper to implement Adam Dunkels' Protothreads with labels as values on large ATmegas with more than 128K flash.

Hint / Warning: This does only work (respectively has to be used) as long as a) there are no 24 bit void pointers and b) the trampolin (+ linker relaxation) hack is not in effect. As of January 2014 both a) and b) are true for arv-gcc — hence do not use this (see TRAMPOLIN_USE).

Parameters

labelAsValue	the target (jump) address' lower 16 bit

Returns

byte 3 of return address or 0

2.4.3.2 uint8_t **addr3Here** (void)

Get the upper (bit 16 ...) part of the current address.

This function is a helper to implement Adam Dunkels' Protothreads with labels as values on large ATmegas with more than 128K flash.

Returns

byte 3 of return address or 0

See also

goto_P
addrHere

2.4.3.3 uint32_t **addrHere** (void)

Get the current address.

This function returns the address from where it was called — or to be exact, the flash address behind the instruction calling this function.

Returns

calling address +2; the number of relevant bits depends on the flash size

2.4.3.4 void **sys1msThread** (void)

The milliseconds system function (or thread)

This thread is scheduled by the runtime very often (high priority). It does work on the condition that one or more ms-ticks occurred since its last schedule. Otherwise it returns / yields at once.

This thread function does

- the basic time keeping for the timed scheduling of other system and user threads,
- the handling of timers with milliseconds resolution,
- the basic process digital / analougue input (DI / AI) handling and filtering,
- some time critical flow / overload control on communication links as well as
- the main polling of the Ethernet driver ENC28J60 every 10 ms.

In normal operation this function should be called at least twice every ms. But if other processor tasks inhibit this it is quite capable to handle several events (= ms ticks) at once. Nevertheless, in case of more than 10 resp. 100 collected ticks the scheduling of user and system threads might come offbeat as well as the sequence of effects of lapsed timers.

In case of more than 250 collected ticks, all might go wrong.

Internal remark: This function is (still) called "...Thread" as it could as well organised as one (as it once was). On the other hand the system 1ms actions have to be handled in one optimised compact action without yielding and blocking. So the (small) Protothreads overhead is skimped at the moment.

2.4.3.5 ptfnct_t sys100msThread (void)

The 100 milliseconds system thread, the function.

This thread is scheduled quite often (high priority). It yields internally until sys100msPeriodFlag is set.

This system thread handles the watchdog and does all periodic timing for the Ethernet stack uIP.

See also

ptSys100msThread

2.4.3.6 ptfnct_t sys1sThread (void)

The 1 second system thread, the function.

This thread is scheduled repeatedly (medium priority). It yields internally or exits until sys1sPeriodFlag is set.

It does all date / time and timer with seconds resolution handling.

See also

ptSys1sThread;

2.4.3.7 void initThreads (void)

Initialise the threads.

Besides some other initialisations this will de-register all user / application threads.

See also

pt.h

2.4.3.8 ptfnct_t outSysInfoThreadF (pt_t ∗ *pt*, FILE ∗ *toStream*)

Output some system (version) info, the thread function.

This thread will output some system info, like

name, version, copyright, build and so on to the standard output. This is meant for system start respectively welcome info. This thread will usually started as child thread (of `pt`) by the macro `runOutSysInfoThread(parent, pt, toStream)` .

As the multi-line output may be longer than `toStream's` output buffer this thread will repeatedly yield until buffer space is available. If started as child thread the parent thread will "inherit" these blocks or waits.

Parameters

pt	the thread state (pointer to raw structure; not NULL)
toStream	streams to switch the standard streams to (if not MULL)

Returns

PT_YIELDED if waiting for conditions that may become automatically true; PT_WAITING if blocked by conditions that other's action may set true; PT_EXITED if stopped by conditions that never will become true (sorry, not all work done and never will; PT_ENDED if ready with all work

2.4.3.9 ptfnct_t outFlashTextThreadF (outFlashTextThr_data_t ∗ *outFlashTextThread,* FILE ∗ *toStream*)

Output a flash string array to standard output thread.

This thread will output a number of strings in flash memory given as an array of pointers to them by initialising the thread with initOutFlashTextThread.

The initialising must be done before starting the thread.

This thread will block (multiple times) until the output buffer has enough space.

If started as child thread the parent thread will "inherit" these blocks or waits as yields.

Parameters

outFlashText-Thread	the thread state data
toStream	streams to switch the standard streams to (if not MULL)

Returns

PT_YIELDED if waiting for conditions that may become automatically true; PT_WAITING if blocked by conditions that other's action may set true; PT_EXITED if stopped by conditions that never will become true (sorry, not all work done and never will; PT_ENDED if ready with all work

2.4.3.10 void initOutFlashTextThread (outFlashTextThr_data_t ∗ *outFlashTextThread,* char const ∗const ∗ *theFlashStrings2out,* uint8_t *noOfFlashStrings2out*)

Initialise the output a flash string array to standard output thread.

After this call the thread outFlashTextThreadF() will (block and) work until `noOfFlashStrings2out` strings were output or a NULL is found in `theFlashStrings2out` which is considered as end of the array.

Parameters

outFlashText-Thread	the thread state data
theFlash-Strings2out	pointer to (flash) array of (flash) strings to be output
noOfFlash-Strings2out	number of (flash) strings to be output

2.4.3.11 p2ptFun registerAppThread (struct mThr_data_t ∗ *thread,* p2ptFun *threadF*)

Register an user / application thread.

Parameters

thread	the thread to register to; must not be NULL.
threadF	the thread function to be registered; NULL means de-register the thread

Returns

the previously registered thread function; NULL if there was none

See also

initSetAppThread(struct mThr_data_t ∗, p2ptFun);

Examples:

main.c.

2.4.3.12 void **initSetAppThread** (struct **mThr_data_t** ∗ *thread,* **p2ptFun** *threadF*)

Register and (force) init an user / application thread.

This function will register `threadF` with `thread` unconditionally and init / reset all. It is meant for first initialisation.

Note

Never call more than once with the same parameters and never call on a started thread. The results may be unwelcome to put it very mildly. If the situation is unclear in that respect, use `registerAppThread(m-Thr_data_t, p2ptFun)` instead.

Parameters

thread	the thread to register to; must not be NULL.
threadF	the thread function to be registered; NULL means de-register the thread

See also

registerAppThread(struct mThr_data_t ∗, p2ptFun)

2.4.4 Variable Documentation

2.4.4.1 uint16_t **minStckP**

Minimal stack pointer value sampled.

System and application software might sample the stackpointer at due points and set this variable to it if larger. A value of 0xFFFF means no (new) samples taken as yet. The recipe to sample the stackpointer `SP` and update its minimal value is

```
uint16_t value = SP;
if (value < minStckP) minStckP = value; // sample stackpointer
```

2.4.4.2 uint8_t **sys100msPeriodFlag**

Flag for system 100ms thread.

This flag will be set by (the time keeping part of) the 1ms thread sys1MsThread.

2.4.4.3 uint8_t **sys1sPeriodFlag**

Flag for system 1s thread.

This flag will be set by (the time keeping part of) the 1ms thread sys1msThread.

Bit 0: "run since" seconds were incremented

Bit 7: "absolute date time" seconds were incremented / adjusted

Bit 6: "absolute date time" seconds were set (heavily changed)

2.5 Timer handling

2.5.1 Overview

Real time automation systems require time keeping functions

- for some use cases as absolute date and time,
- timers toward a future (absolute) date and local time
- more often timers relative to "now" and
- usually by providing cyclic applications threads scheduled (/signaled / flagged) at constant frequencies.

weAutSys offers all those possibilities. For the aspect of date and time see time keeping.

Timer functions relative to start or now can easily (and under some conditions better) be implemented by counting in the cyclic application threads provided the timing period is a small multiple of the task's or cycle's period.

For all (other) timer purposes weAutSys offers any number of timers. There are three types of system supported timers

- 1 ms resolution, duration / interval type, max. period 24 days
- 1 s resolution, duration / interval type, max. period 68 years
- 1 s resolution, date / time oriented, max. period 68 years

All three timer types are quite efficiently and uniformly handled by the runtime. The permanent system load is not dependent on the number of running timers (it is O(1)).

The speciality of the third type (date / time oriented) is that any adjustments or corrections of the system's date & time will (effectively) change the current endTime of running timers of that type by the same amount. Hence, that a timer of that type started with a period aiming at "next week Monday 10:23:11" will lapse at that time even if clock corrections (by NTP, e.g.) or summertime shifts occurred in between.

The other two timer types will strictly adhere to their set period (or frequency).

2.5.2 Warnings

- Conditions and rules described anywhere in this documentation as mandatory are essential for the whole system's health.
- Modifying variables described as "do not touch" or "read only", like timing values, timer states, linked list pointers etc., directly by application software almost certainly breaks essential system invariants.
- Too long periods of non yielding CPU usage is deadly for a non preemptive system, see Rules for user threads' CPU usage.

  And spin waiting is (almost) always a crime here. Do use the yield / wait while / until thread constructs instead.

Data Structures

- struct timer_t

  The timer data type resp.

Defines

- #define ms_TIMER_TYPE

  A milliseconds based timer.
- #define MSEC_DAY 86400000

  Milliseconds per day.
- #define MSEC_HOUR 3600000

  Milliseconds per hour.
- #define MSEC_MINUTE 60000

  Milliseconds per minute.
- #define MSEC_WEEK

  Milliseconds per week.
- #define sec_dat_TIMER_TYPE

  A date / time oriented timer with seconds resolution.
- #define sec_dur_TIMER_TYPE

  A duration / period oriented timer with seconds resolution.
- #define SECONDS_IN_DAY 86400

  Seconds in a day.
- #define SECONDS_IN_WEEK

  Seconds in a week.
- #define SECONDS_IN_YEAR

  Seconds in a (normal) year.
- #define TIMER_LAPSED 0x00

  Timer lapsed.
- #define TIMER_RUNNING

  Timer running.
- #define TIMER_STATE_MASK

  upper 4 bits of timer.state (the state)
- #define TIMER_STOPPED

  Timer stopped.
- #define TIMER_TYPE_DEFAULT

  Set type to default or leave untouched.
- #define TIMER_TYPE_FREE

  Timer free.
- #define TIMER_TYPE_MASK

  lower 4 (2) bits of timer.state (the type)
- #define TIMER_UNUSED

  Timer unused.
- #define timerLapsed(timer)

  Is a timer lapsed.
- #define timerRunning(timer)

  Is a timer running.

Functions

- void freeTimer (struct timer_t ∗timer)

  Free a timer.
- void haltTimer (struct timer_t ∗timer)

  Halt a timer.
- uint8_t initTimer (struct timer_t ∗timer, uint32_t period, uint8_t type)

  Initialise a timer.

- uint32_t msClock (void) __attribute__((always_inline))

  Milliseconds since power up.
- void pauseTimer (struct timer_t ∗timer)

  Stop / pause a timer.
- struct timer_t ∗ removeTimerFromList (const struct timer_t ∗timer, struct timer_t ∗list)

  Remove a timer from a list.
- void restartTimer (struct timer_t ∗timer)

  Start a timer with its own interval from now.
- void startNextInterval (struct timer_t ∗timer)

  Start a timer's next interval.
- void startTimer (struct timer_t ∗timer, uint32_t period, uint8_t type)

  Start a timer with interval / type from now.
- void startTimerAbs (struct timer_t ∗timer, uint32_t endTime, uint8_t type)

  Start a timer (with absolute end time)

Variables

- uint16_t msAbsClockCount

  The system ms clock value for local time.
- uint16_t msSystClockCount

  The system ms counter / clock value for running / system up time.
- struct timer_t msTimers

  The system milliseconds time and timers.
- struct timer_t secDatTimers

  The system seconds resolution date / time oriented timers.
- struct timer_t secDurTimers

  The system seconds resolution duration oriented timers.
- struct timer_t ∗ timerLists [ ]

  The list of timer lists.

2.5.3 Define Documentation

2.5.3.1 #define MSEC_MINUTE 60000

Milliseconds per minute.

See also

MSEC_HOUR

2.5.3.2 #define MSEC_HOUR 3600000

Milliseconds per hour.

See also

MSEC_MINUTE

2.5.3.3 #define MSEC_DAY 86400000

Milliseconds per day.

This value can be used as milliseconds timer period for every day repetitive actions like closing or opening windows, shutters or doors.

Note

Keep in mind that milliseconds timers will fail for periods longer than 28 days. Consider using seconds based timers.

See also

MSEC_MINUTE

2.5.3.4 #define MSEC_WEEK

Milliseconds per week.

This value can be used as period for milliseconds timers of one week. Often unused aggregates, like pumps, are run for a short time every week to be kept movable.

Note

Keep in mind that milliseconds timers will fail for periods longer than 4 weeks. Consider using seconds based timers.

See also

MSEC_MINUTE
SECONDS_IN_WEEK

2.5.3.5 #define SECONDS_IN_DAY 86400

Seconds in a day.

This is the divisor used to split second based absolute time into a) days since base date and b) seconds since midnight.

This value can be used as period for seconds based timers.

2.5.3.6 #define SECONDS_IN_WEEK

Seconds in a week.

This value can be used as period for seconds based timers.

Often unused aggregates, like pumps, are run for a short time every week to be kept movable.

2.5.3.7 #define SECONDS_IN_YEAR

Seconds in a (normal) year.

This value is the equvalent of 365 days.

2.5.3.8 #define ms_TIMER_TYPE

A milliseconds based timer.

This is a timer with milliseconds resolution that is duration / period / frequency oriented. Its endTime relates to msClock.

Examples:

main.c.

2.5.3.9 #define sec_dur_TIMER_TYPE

A duration / period oriented timer with seconds resolution.

This is a timer with seconds resolution that is duration / period oriented. Its endTime relates to secClock.

2.5.3.10 #define sec_dat_TIMER_TYPE

A date / time oriented timer with seconds resolution.

This is a timer with seconds resolution that is date / time oriented.

Its endTime relates to secClock. In effect the endtimes of running timers of that type are adjusted by the the same amount if any date and time settings or adjustments should occur.

So, for example, a timer of this type started with SECONDS_IN_WEEK duration an Monday 7:15, will lapse next Monday 7:15 no matter what adjustments (by NTP client e.g.) or even summertime shifts were made during that week. On the other hand big adjustments (like setting date and time to totally different values) may bring a timer of this type far into future or let it lapse at once.

2.5.3.11 #define TIMER_TYPE_FREE

Timer free.

This is the type of a timer that is neither used nor owned by any task. If used as type value for (re-) start, this acts as default.

If the user / application software chooses to have a pool of timers (en lieu de one static timer for every purpose) it will have to handle the transitions to / from this state.

See also

TIMER_LAPSED
TIMER_STATE_MASK
initTimer()

2.5.3.12 #define TIMER_LAPSED 0x00

Timer lapsed.

This is the state value of a timer that has run to the end of its (actual) period, i.e. lapsed or expired.

See also

TIMER_RUNNING
TIMER_STOPPED
TIMER_UNUSED
TIMER_STATE_MASK

2.5.3.13 #define TIMER_RUNNING

Timer running.

This is the state value of a timer that is running and not yet lapsed nor stopped.

Note

This state is equivalent of being held in a system's list of running timers and must never be modified directly.

See also

TIMER_LAPSED
TIMER_TYPE_FREE
TIMER_STATE_MASK

2.5.3.14 #define TIMER_STOPPED

Timer stopped.

This is the state value of a timer that has been stopped. Its period value will be set to the remaining rest time; it will have to be restored to its original value before a timer restart.

This usually makes no sense for date time oriented timers

See also

TIMER_LAPSED
TIMER_STATE_MASK

2.5.3.15 #define TIMER_UNUSED

Timer unused.

This is the state value of a timer that is currently unused but still owned by a application or system task. This owner is responsible for re-using or freeing that timer.

See also

TIMER_LAPSED
TIMER_STATE_MASK

2.5.3.16 #define timerLapsed(*timer*)

Is a timer lapsed.

This will evaluate to true if the timer was started and has run to its end.

Parameters

timer	The timer to be asked (never NULL !)

See also

timerRunning

Examples:

main.c.

2.5.3.17 #define timerRunning(*timer*)

Is a timer running.

This will evaluate to true if the timer is still running; i.e. has not yet run to its end.

Note

timerLapsed() and !timerRunning are not the same.

Parameters

timer	The timer to be asked (never NULL !)

See also

timerLapsed

2.5.4 Function Documentation

2.5.4.1 uint32_t msClock (void)

Milliseconds since power up.

The value returned is the basis of all weAutSys (relative) timing in milliseconds resolution. It is the 32 bit unsigned number of milliseconds since system start or reset. The value will wrap around after a bit more than 49 days.

A modulo arithmetic compare function (geU32ModAr()) used for timers / time values of that type will give an correct >= answer even over that wrap around provided the the difference is less than half the above range.

Hence timers for about 24,8 days can be made with milliseconds resolution.

Monotony in the sense of modulo 2∗∗32 (or geU32ModAr()) is guaranteed, but there may be gaps (little leaps).

See also

ucnt32_t

2.5.4.2 struct timer_t∗ removeTimerFromList (const struct timer_t ∗ *timer*, struct timer_t ∗ *list*) `[read]`

Remove a timer from a list.

This function removes the `timer` from the `list` and then sets `timer->next` to NULL. The return value will point to the element that was `timer's` predecessor in `list`.

If `timer` is not found in `list` NULL is returned and neither `timer` nor `list` will be touched.

Note

No parameter must be NULL !
This is a (system) helper function only. No checks are made if the call is legal or makes sense.

Parameters

timer	the timer to be removed from list
list	the list head, but may be any timer (in a linked list) the search for timer to be removed will start after

2.5.4.3 void **startTimer** (struct **timer_t** * *timer,* uint32_t *period,* uint8_t *type*)

Start a timer with interval / type from now.

If timer is NULL or it is free nothing will happen.

If the `period` is set to 0 the state will be set to unused (n.b. not to lapsed) and nothing else will happen.

Otherwise this function will set the `timer` with `period` ms or s running from now, no matter what its before state has been.

Parameters

timer	points to the timer to be started
period	is the next interval duration in s or ms depending on `type`
type	may be ms_TIMER_TYPE, sec_dur_TIMER_TYPE or sec_dat_TIMER_TYPE; the upper 6 state bits are ignored

2.5.4.4 void **restartTimer** (struct **timer_t** * *timer*)

Start a timer with its own interval from now.

If timer is NULL or it is free nothing will happen.

If its `period` is 0 the state will be set to unused and nothing else will happen.

Otherwise this function will set the `timer` with its own `period` running from now, no matter what its before state has been.

Parameters

timer	points to the timer to be (re-) started

Examples:

main.c.

2.5.4.5 void **startTimerAbs** (struct **timer_t** * *timer,* uint32_t *endTime,* uint8_t *type*)

Start a timer (with absolute end time)

If timer is NULL or it is free nothing will happen.

If the `endTime` set is not in the future the state will be set to unused (n.b. not to lapsed) and nothing else will happen.

Otherwise this function will set the `timer` running to the `endTime` (of `type`).

Parameters

timer	points to the timer to be started
endTime	is the next interval's end in s or ms depending on `type`
type	may be ms_TIMER_TYPE, sec_dur_TIMER_TYPE or sec_dat_TIMER_TYPE; the upper 6 state bits are ignored

2.5.4.6 void **startNextInterval** (struct **timer_t** ∗ *timer*)

Start a timer's next interval.

If timer is NULL or it is free nothing will happen.

If the `period` is 0 nothing will happen, also.

Otherwise this function starts or restarts the `timer` with its actually set internal `period` from its actual end time.

This actual end time may be in future for a running timer. In this case the timer's next lapse will be postponed farther in the future.

The actual end time may be now or in the past for a lapsed timer. In this case it is a "frequency exact" restart, while startTimer() would be used for an "absolute time exact" restart.

Attention: If the the timers actual end time so far in the past that the prolonged end time is in the past too, the timer will not be re-started. It will be lapsed at once.

If the timer is in state halted/paused or unused it will start with its internal settings from now.

Parameters

timer	points to the timer to be re-started

Examples:

main.c.

2.5.4.7 void **haltTimer** (struct **timer_t** ∗ *timer*)

Halt a timer.

If timer is NULL or it is free nothing will happen.

Just stop the timer. This will set the timer to state `TIMER_UNUSED`.

Parameters

timer	points to the timer to be halted

2.5.4.8 uint8_t **initTimer** (struct **timer_t** ∗ *timer,* uint32_t *period,* uint8_t *type*)

Initialise a timer.

This will set the timer to state unused and to the given `period` and `type` non-regarding its current state.

This function should also be called by the owner of a (static!) timer for first time (reset) initialisation. If the user / application software chooses to have a pool of N timers (probably as static timer_t tPool[N]) this function should be called for every element.

If `timer` is NULL nothing will be done; if its state is TIMER_RUNNING it will be removed from the system's list of running timers.

See also

freeTimer

Parameters

timer	points to the timer to be initialised
period	the duration in s or ms dependinf on `type`
type	may be ms_TIMER_TYPE, sec_dur_TIMER_TYPE or sec_dat_TIMER_TYPE; the upper 4 state bits are ignored

Returns

false (0) if the previous state of `timer` was not free, otherwise, as a signal to user software that chose to handle (a list of) free timers, true (!=0)

Examples:

main.c.

2.5.4.9 void pauseTimer (struct timer_t * *timer*)

Stop / pause a timer.

If timer is NULL, unused or free nothing will happen.

A bit like `haltTimer(struct timer_t)`, but the internal `period` will be reduced to the actual run duration not yet elapsed and the state will be TIMER_STOPPED.

Parameters

timer	pointer to the timer to be freed

2.5.4.10 void freeTimer (struct timer_t * *timer*)

Free a timer.

This function sets a timer in the type TIMER_TYPE_FREE.

If `timer` is NULL nothing will be done; if it is running it will be removed from the system's list of running timers.

See also

initTimer

Parameters

timer	pointer to the timer to be freed

2.5.5 Variable Documentation

2.5.5.1 uint16_t msSystClockCount

The system ms counter / clock value for running / system up time.

The value is in the range 0..999 and will be incremented respectively wrapped around modulo 1000 by system software (i.e. in the system 1ms thread). Wrapping means incrementing the secClock() (i.e. secDurTimers.end.-v32).

Monotony (in modulo 1000 terms) is guaranteed, but there may be gaps.

This value is not affected by absolute (local) date / time settings or corrections.

Never to be modified elsewhere.

See also

msAbsClockCount

2.5.5.2 uint16_t msAbsClockCount

The system ms clock value for local time.

The value is in the range 0..999 and will be incremented respectively wrapped around modulo 1000 by system software (i.e. in the system 1ms thread). Wrapping means incrementing all seconds resolution clock values, like secTime308Loc() etc., but not secClock().

Monotony (in modulo 1000 terms) is guaranteed except for time / date settings corrections.

Never to be modified elsewhere.

See also

msSystClockCount

2.5.5.3 struct timer_t msTimers

The system milliseconds time and timers.

This is (a pseudo timer) for system internal timing and list handling use only!

.next is the root of the system's internal ordered linked list of active / running ms based timers.

.end is the (one) system ms 32 bit counter.

.period is the time zone offset in seconds (0 for WET, 3600 for CET, 7200 for CEST [EU DST rules implied]; -5 for EST .. -8 PST [Northamerican DST rules]).

Application software must not modify any field.

2.5.5.4 struct timer_t secDatTimers

The system seconds resolution date / time oriented timers.

This is (a pseudo timer) for system internal timing and list handling use only!

.next is the root of the system's internal ordered linked list of active / running seconds based timers.

`secDatTimers.end` is the local time in seconds since March 1st 2008 UTC.

`secDatTimers.period` is the (`msTimers.end` stamp) value of the absolute date / time last adjustment or setting.

Application software must not modify any field.

2.5.5.5 struct timer_t secDurTimers

The system seconds resolution duration oriented timers.

This is (a pseudo timer) for system internal timing and list handling use only!

.next is the root of the system's internal ordered linked list of active / running seconds based timers.

`secDurTimers.end` is the (one) system seconds clock counting the seconds since last reset / re-start.

`secDurTimers.period` is the system start time in seconds since March 1st 2008 UTC.

Application software must not modify any field.

2.5.5.6 struct timer_t* timerLists[ ]

The list of timer lists.

Index is the type: ms_TIMER_TYPE sec_dat_TIMER_TYPE sec_dur_TIMER_TYPE ([0..2])

See also

TIMER_TYPE_MASK

2.6 Time keeping (date and time)

2.6.1 Overview

Real time automation systems require time keeping functions

- for some use cases as absolute date and time,
- timers toward a future (absolute) date and local time
- more often timers relative to "now" and
- usually by providing cyclic applications threads scheduled (/signalled / flagged) at constant frequencies.

weAutSys offers all those possibilities. (For the aspect of timers see timer handling.) Its basic timing is done in milliseconds (ms) resolution and by counting ticks from start or system reset in a 32 bit internal counter. The absolute timing is done in seconds (s) resolution in terms of seconds since March 1 2008 00:00 UTC.

Some timing functions, values and types are defined for weAutSys' internals. But all may be exploited by application software.

2.6.2 Milliseconds timing

weAutSys internal basic timing is milliseconds based. The milliseconds since system start / reset are counted as an unsigned 32 bit number.

All periodic tasking for automation cycles and all timers with milliseconds resolution is based on that "milliseconds since start" time. It will run on until power down or next reset and never be manipulated (adjusted) by any system operation.

The 32 bit unsigned number of milliseconds since system start will wrap around after a bit more than 49 days. This will limit the maximum period or runtime of weAutSys's timers with millisecond resolution to half that period; that is 24,8 days or 2.147.483.647 ms. Hence, for longer periods than three weeks use the timers with seconds resolution — a bearable restriction in most use cases.

2.6.3 Timing with seconds resolution

weAutSys's relative timing is based on having the seconds since system start continuously counted. This (32 bit unsigned) value will wrap around in about 136 years giving about 68 years (unproblematic) timer periods. (At present we don't care on longer timers and do not expect a device running that long without reset or power down — but that may be too pessimistic.)

Timers with seconds (s) resolution that are duration / period oriented are based on this absolute time.

2.6.4 Date time oriented timing with seconds resolution

weAutSys's absolute seconds timing is measured as seconds since March 1 2008 00:00 UTC (as a 32 bit unsigned value). The offset from seconds since start is, of course, basically the systems reset / restart time.

To get the local time and date two further offsets are handled:

- a (32 bit signed) time zone offset (usually 1h multiples and + is east of Greenwich) and
- an summer time offset (usually 0 or +1h)

The time zone offset is best be set by the answer of an appropriate DHCP server. Using one DHCP server + the NTP server provided by this DHCP has the advantage of automatically having the same settings on the whole automation subnet.

Switching DST on and off is based on the currently valid EU or Northern America's summer time rules.

The reset time offset will be known at the first incoming date / time information (best by weAutSys's built in NTP client and a NTP server made known by DHCP). Till this first setting it will be preset by the software build time, and hence be more or less in the past. Applications dependent on date and time should check if this (first) time setting has occurred.

The runtime's absolute "zero date" (March 1 2008 00:00 UTC) is 1204329600s in UNIX (1.1.1970 based) and (approximately) 3413318400s in NTP (1.1.1900 based) time. So there is just a simple offset to usual absolute time informations. The problem of leap seconds is not yet handled (but may be in future).

The wrap around of that absolute seconds time (in the year 2144) is in no way dealt with but the NTP wrap around in 2136 is.

All date time related timers with seconds (s) resolution are based on this absolute time.

Files

- file timing.h

  weAutSys definition of timing services

Data Structures

- struct datdur_t

  "Time since" as a structure: a duration or a date/time
- struct date_t

  Date structure: a date in our time.
- struct dst_rule_year_t

  Rule structure: DST rules for a given (set of) year(s)

Defines

- #define DEFAULT_START_TIME

  Reset (or default) start time.
- #define MARCH_2008_NTP 3413318400

  March 2008 as NTP seconds.
- #define MARCH_2008_UNIX 1204329600

  March 2008 as UNIX seconds.
- #define OFFSET2NTPTIME

  same as MARCH_2008_NTP
- #define OFFSET2UNIXTIME

  same as MARCH_2008_UNIX
- #define PRESC_TRIM_FAC

  Prescaled time factor to millisecond.
- #define secTime308Loc(x)

  Seconds since March 1st 2008 local time.
- #define startTime308UTC

  The reset / start time in seconds since March 1st 2008 UTC.
- #define VCO_DEFAULT 7

  Trimming of timing oscillator.
- #define zoneOffsetSec

  The time zone offset in seconds.
- #define zoneOffsetZone

  zoneOffsetSec's byte [1]

Date calculation defines

- #define DAYS_IN_YEAR

  Days in a year.
- #define DAYS_IN4YEARS

  Days in four years.
- #define APRIL_OFF

  Offset of April.
- #define MAY_OFF

  Offset of May.
- #define JUNE_OFF

  Offset of June.
- #define JULY_OFF

  Offset of July.
- #define AUGUST_OFF

  Offset of August.
- #define SEPTEMBER_OFF

  Offset of September.
- #define OCTOBER_OFF

  Offset of October.
- #define NOVEMBER_OFF

  Offset of November.
- #define DECEMBER_OFF

  Offset of December.
- #define JANUARY_OFF

  Offset of (next) January.
- #define FEBRUARY_OFF

  Offset of (next) February.
- #define MARCH_2012

  March 2012.
- #define MARCH_2016

  March 2016.
- #define MARCH_2020

  March 2020.
- #define MARCH_2100

  March 2100.

Date handling functions

- uint8_t setDatByDays (date_t ∗datStr, uint16_t ds)

  Set date structure by days since since March 2008.
- uint16_t getDaysByDat (date_t ∗datStr)

  Get days since since March 2008 by date structure.
- uint8_t getMarchYearByDays (uint16_t ∗daysInYear, uint16_t ds)

  Get year starting March (includes next January and February)

Functions

- uint8_t cnt12u8_8 (void) __attribute__((pure))

  Get the 12.8 (16) µs tick value (8 bit)
- uint32_t datdur2sec (datdur_t ∗timStr)

  Calculate seconds from a datdur_t structure.
- const dst_rule_year_t ∗ getDSTrule (uint8_t year)

  Get the EU, US &c DST rule data for a given year.
- uint32_t getFATtime (void)

  The local time as FAT time.
- uint8_t getVCOsetting (void) __attribute__((pure))

  Trimming of timing oscillator.
- uint8_t isEUdstSwitchDay (date_t ∗datStr)

  Is date represented in date structure EU DST switching day.
- void sec2datdur (datdur_t ∗timStr, uint32_t timSec)

  Convert seconds to a datdur_t structure.
- uint32_t secClock (void) __attribute__((always_inline))

  The system's run time in seconds.
- uint32_t secTime308UTC (void) __attribute__((always_inline))

  Seconds since March 1st 2008 UTC.
- uint32_t secTimeNtpUTC (uint32_t secTime308UTC) __attribute__((always_inline))

  Convert to seconds since January 1st 1990 UTC (NTP time)
- uint32_t secTimeUnixUTC (uint32_t secTime308UTC) __attribute__((always_inline))

  Convert to seconds since January 1st 1970 UTC (Unix time)
- uint8_t setDST (uint8_t dlt)

  Set if current time is DST.
- void setVCOnormal (void) __attribute__((always_inline))

  Trimming of timing oscillator.
- uint8_t setZoneOffsetSec (uint32_t newOffset)

  Adjust / set zone offset.
- void slowVCOdown (void) __attribute__((always_inline))

  Trimming of timing oscillator.
- void speedVCOup (void) __attribute__((always_inline))

  Trimming of timing oscillator.

Variables

- uint8_t adjustUTCcount

  Count of secTime308Loc adjustments.
- uint8_t cn12u8

  The 12,8 µs counter summand.
- uint32_t combinedOffset

  The combined offset (local - UTC)
- uint8_t isDST

  Is current time DST.
- uint8_t msIntTick

  The ms tick counter.

2.6.5 Define Documentation

2.6.5.1 #define PRESC_TRIM_FAC

Prescaled time factor to millisecond.

By this factor the processor clock period pre-scaled by 256 will be the best <= approach to one millisecond.

For a 20 MHz machine the pre-sclaled clock period will be 12.8 µs and this factor 78. That will yield 998.4 µs which cal easily be slowed down by VCO to 1ms.

On a 16 MHz machine that will be 16µs, 62 and 992µs

2.6.5.2 #define VCO_DEFAULT 7

Trimming of timing oscillator.

The higher the value the faster the oscillator for the system 1 ms ticks will be running (in the mean). The relation is non-linear.

Background: This value is used to slow down respectively lengthen every 1st .. 255th millisecond tick period a tiny bit. All others will be a wee bit too fast.

Range: 1 .. 255 (0 would act as 256)

The best value for a 16 MHz machine is 2. The best setting for weAut_01 board with a good 20 MHz Quartz for ATmega1284's processor clock is 7.

2.6.5.3 #define secTime308Loc(*x*)

Seconds since March 1st 2008 local time.

This gives the local time in seconds since (Saturday) March 1st 2008 in local time — i.e. with zone offsets and summertime (DST) shifts.

See also

secClock()
msAbsClockCount

2.6.5.4 #define startTime308UTC

The reset / start time in seconds since March 1st 2008 UTC.

This is the systems reset time in seconds in seconds since weAutSys's zero date March 1st 2008 UTC. It will be initialised by DEFAULT_START_TIME and be adjusted by the first setting of time and date (e.g. by NTP client function) after reset.

2.6.5.5 #define zoneOffsetSec

The time zone offset in seconds.

This is usually in multiples of hours (3600) and in the range of -12..+12 ($*$3600) — positive being east of Greenwich.

The value should usually set by DHCP, the reset / default is 3600, i.e. CET.

2.6.5.6 #define DEFAULT_START_TIME

Reset (or default) start time.

This absolute time in seconds since March 1st 2008 that will usually be set as last compile time (in the file maketime-_utc.h by utc_maketime.bat).

This time will be used as reset value for the internal seconds clock.

Of course it will be quite wrong (i.e. more or less in past) until any adjustments by human command (CLI), time server answers or else are available.

`DEFAULT_START_TIME` can be used as date and time of make but will usually differ from SYST_DAT, which is the "official" (sub) version date.

2.6.5.7 #define MARCH_2008_NTP 3413318400

March 2008 as NTP seconds.

This is the NTP date for March 1 2008 00:00 UTC.

In weAutSys's seconds based absolute time this is the zero date.

2.6.5.8 #define MARCH_2008_UNIX 1204329600

March 2008 as UNIX seconds.

This is the UNIX date for March 1 2008 00:00 UTC (weAutSys's zero date).

See also

MARCH_2008_NTP

2.6.5.9 #define DAYS_IN_YEAR

Days in a year.

It is of course 365; a leap year will be 1 day longer.

This leap day, the 29th of February (29.2.leapYear), would better have been put at the end of the year by the popes. That is in effect done so by clever calendar algorithms (not invented here but heavily used in weAutSys).

2.6.5.10 #define DAYS_IN4YEARS

Days in four years.

It is of course $4 * 365 + 1$.

2.6.5.11 #define APRIL_OFF

Offset of April.

Days from March 1st to April 1st (same year).

2.6.5.12 #define MAY_OFF

Offset of May.

Days from March 1st to May 1st (same year).

2.6.5.13 #define JUNE_OFF

Offset of June.

Days from March 1st to June 1st (same year).

2.6.5.14 #define JULY_OFF

Offset of July.

See also

JUNE_OFF

2.6.5.15 #define AUGUST_OFF

Offset of August.

See also

JUNE_OFF

2.6.5.16 #define SEPTEMBER_OFF

Offset of September.

See also

JUNE_OFF

2.6.5.17 #define OCTOBER_OFF

Offset of October.

See also

JUNE_OFF

2.6.5.18 #define NOVEMBER_OFF

Offset of November.

See also

JUNE_OFF

2.6.5.19 #define DECEMBER_OFF

Offset of December.

Days from March 1st to December 1st (same year).

See also

JUNE_OFF

2.6.5.20 #define JANUARY_OFF

Offset of (next) January.

Days from March 1st to January 1st (next year).

See also

DECEMBER_OFF

2.6.5.21 #define FEBRUARY_OFF

Offset of (next) February.

Days from March 1st to February 1st (next year).

2.6.5.22 #define MARCH_2012

March 2012.

This is the number of days at march 1st 2012 (1.3.2012 / 2012-03-01) counted since March 1st 2008 (1.3.2008 / 2008-03-01).

2.6.5.23 #define MARCH_2016

March 2016.

This is the number of days at march 1st 2016 (1.3.2016 / 2016-03-01) counted since March 1st 2008 (1.3.2008 / 2008-03-01).

2.6.5.24 #define MARCH_2020

March 2020.

This is the number of days at march 1st 2020 (1.3.2020 / 2020-03-01) counted since March 1st 2008 (1.3.2008 / 2008-03-01).

2.6.5.25 #define MARCH_2100

March 2100.

This is the number of days at march 1st 2100 (1.3.2100 / 2100-03-01) counted since March 1st 2008 (1.3.2008 / 2008-03-01).

2.6.6 Function Documentation

2.6.6.1 void setVCOnormal (void)

Trimming of timing oscillator.

This function will set the "value controlled oscillator" for the system's 1 ms tick to normal (usually best) speed.

Note

If weAutSys's NTP client is active this function is for its exclusive use only.

See also

VCO_DEFAULT

2.6.6.2 uint8_t getVCOsetting (void)

Trimming of timing oscillator.

This function returns the current speed of the "value controlled oscillator" for the system's 1 ms tick. VCO_DEFAULT is usually normal / best setting; higher values are faster.

2.6.6.3 void speedVCOup (void)

Trimming of timing oscillator.

This function will speed up the "value controlled oscillator" for the system's 1 ms tick a little bit. This is for catching up quite little differences to an externally given absolute (NTP e.g.) time.

Little differences in that sense are below -200 ms.

See the note at setVCOnormal.

See also

VCO_DEFAULT

2.6.6.4 void slowVCOdown (void)

Trimming of timing oscillator.

This function will slow down the "value controlled oscillator" for the system's 1 ms tick a little bit. This is for catching up quite little differences to an externally given absolute (NTP e.g.) time.

Little differences in that sense are below +100 ms.

See the note at setVCOnormal.

See also

VCO_DEFAULT
setVCOnormal

2.6.6.5 uint8_t cnt12u8_8 (void)

Get the 12.8 (16) µs tick value (8 bit)

This function returns the actual 12.8 µs respectively 16 µs tick count. This value will wrap about every 3.27 ms on a 20 MHz machine and every 4 ms at 16 MHz. One tick (increment) is 256 processor clocks.

It might be used for quite exact measures of software execution times the normal milliseconds resolution is of little use for.

Hint: 12.8 µs is correct for a 20 MHz machine. For slower machines the counted period is proportionally longer. For 16 MHz it's 16 µs.

See also

PRESC_TRIM_FAC

2.6.6.6 uint32_t secClock (void)

The system's run time in seconds.

The value will be handled (incremented) by system software (i.e. the system's 1s thread). Monotony is guaranteed. There may be gaps, but that is very improbable — all miss-behaving (application) software leading to this should get a watchdog reset.

Do not modify the underlying variable! See also the warning.

See also

msClock(void)
msSystClockCount

2.6.6.7 uint32_t getFATtime (void)

The local time as FAT time.

The local time contains zone offsets and summertime (DST) shifts. This function returns this time in a 32 bit (uint32_t) packed structure used for the traditional FAT file systems. Note that this (old DOS) time format has a 2s resolution only. The format is:

bit 31..25: year from 1980 (0..127; 2012 is 32)

bit 24..21: month (1..12)

bit 20..16: day in month (1..31)

bit 15..11: hour (0..23)

bit 10...5: minute (0..59)

bit 4...0: second / 2 (0..29, 30 if leap second)

See also

secTime308Loc

2.6.6.8 uint32_t secTime308UTC (void)

Seconds since March 1st 2008 UTC.

The value will be the time in seconds since weAutSys's zero date March 1st 2008 UTC.

UTC here means without any zone or DST offsets.

The returned value can easily be converted to UNIX or NTP time.

See also

secTime308Loc()

2.6.6.9 uint32_t secTimeUnixUTC (uint32_t *secTime308UTC*)

Convert to seconds since January 1st 1970 UTC (Unix time)

The returned value will be time in seconds since the UNIX time start date.

Parameters

secTime308UTC	the time in seconds since March 2008 UTC

See also

secClock()
secTime308UTC()

2.6.6.10 uint32_t **secTimeNtpUTC** (uint32_t *secTime308UTC*)

Convert to seconds since January 1st 1990 UTC (NTP time)

The returned value is the time in seconds since the NTP time start date. Called with secClock() as parameter `stamp` this value would be used as a NTP server respectively indirectly adjusted by the NTP client function.

Parameters

secTime308UTC	the time in seconds since March 2008 UTC

See also

secClock()
secTime308UTC()

2.6.6.11 uint8_t **setDST** (uint8_t *dlt*)

Set if current time is DST.

Returns

0 if no changes were made, otherwise 1

See also

isDST

2.6.6.12 uint8_t **setZoneOffsetSec** (uint32_t *newOffset*)

Adjust / set zone offset.

This function sets zoneOffsetSec and handles all side effects in the case of a change.

Returns

true if changed

2.6.6.13 void **sec2datdur** (**datdur_t** ∗ *timStr,* uint32_t *timSec*)

Convert seconds to a datdur_t structure.

This function converts a 32 bit unsigned seconds value to a time structure of type datdur_t. Its interpretation as duration or absolute date and time depends on the parameter's semantic.

Hint: This is a quite expensive operation for an 8 bit RISC machine without divide instruction. The result `timStr` should be kept if usable more than once.

Parameters

timStr	pointer to the time structure to be set
timSec	the time in seconds

See also

datdur2sec

2.6.6.14 uint32_t datdur2sec (datdur_t ∗ *timStr*)

Calculate seconds from a datdur_t structure.

This function converts the content of a structure of type datdur_t to the corresponding 32 bit unsigned seconds value. It is the inverse of sec2datdur().

Hint: This function is not quite cheap for an 8 bit RISC machine with just a 8bit ∗ 8bit = 16bit multiply instruction. The returned result should be kept if needed more than once.

Parameters

timStr	pointer to the time structure to be used

Returns

the seconds calculated from timStr

2.6.6.15 uint8_t setDatByDays (date_t ∗ *datStr*, uint16_t *ds*)

Set date structure by days since since March 2008.

This function sets the (16 bit unsigned) number of days since March 2008 into the structure datStr and all its other fields.

Parameters

datStr	pointer to the date structure to be set
ds	days since weAutSys's day 0

Returns

the resulting weekday (1..7) or 0 in erroneous situations

See also

datdur2sec
getDaysByDat

2.6.6.16 uint16_t getDaysByDat (date_t ∗ *datStr*)

Get days since since March 2008 by date structure.

This calculates the (16 bit unsigned) number of days since March 2008 from the structure and returns the result. All fields of datStr except day of week (ed) are used and must have correct and consistent values.

Before March 2008 0 is returned as is for everything after about ∼2177.

Parameters

datStr	pointer to the date structure to be used

Returns

days since weAutSys's day 0

See also

setDatByDays
datdur2sec

2.6.6.17 uint8_t getMarchYearByDays (uint16_t ∗ *daysInYear,* uint16_t *ds*)

Get year starting March (includes next January and February)

Parameters

ds	days since weAutSys's day 0
daysInYear	pointer to extra result: the the days in the respective year (0 = March 1st); if NULL this information is lost

Returns

the resulting year 8 .. = 2008 ..

See also

datdur2sec

2.6.6.18 const dst_rule_year_t∗ getDSTrule (uint8_t *year*)

Get the EU, US &c DST rule data for a given year.

This function returns a pointer to the DST rule structure for the year requested in flash memory (!). Use pgm_read_byte() respectively pgm_read_dword() accordingly to get single elements. Or — to have the whole struture in RAM use

```
dst_rule_year_t currentRuleValues;
memcpy_P(&currentRuleValues, getDSTrule(year), sizeof(dst_rule_year_t));
```

Parameters

year	0..255 interpreted as 2000..2255

2.6.6.19 uint8_t isEUdstSwitchDay (date_t ∗ *datStr*)

Is date represented in date structure EU DST switching day.

This function checks if the date is either the last Sunday in March (returns 3) or the last Sunday in October (returns 10) or non of both (returns 0).

Parameters

datStr	pointer to the date structure

Returns

the resulting 3, 10 or 0

2.6.7 Variable Documentation

2.6.7.1 uint8_t msIntTick

The ms tick counter.

This variable is written, i.e. incremented, in the tick interrupt handler `ISR(TIMER0_COMPA_vect)` only.

This variable must never (!) be modified elsewhere.

Being just one byte reading it is atomic.

See also

syst_threads.h

2.6.7.2 uint8_t cn12u8

The 12,8 µs counter summand.

Incremented in the 1ms interrupt by PRESC_TRIM_FAC (or PRESC_TRIM_FAC + 1 if VCO slow).

This variable is not to be used directly by user / application software. Use it idirectly by cnt12u8_8().

2.6.7.3 uint8_t adjustUTCcount

Count of secTime308Loc adjustments.

Significant local time adjustments are counted here. 0 means no time date setting since reset (hence usually incorrect).

Counting wraps from 255 to 1.

Significant means hard setting of secTime308Loc or its ms (0.999) fraction part msAbsClockCount. Adjustments by variable controlled oscillator (VCO) are not counted.

2.6.7.4 uint32_t combinedOffset

The combined offset (local - UTC)

This is the difference from local time to UTC.

In other words it is the sum zoneOffsetSec + DST offset

Do never ever modify this value directly. That may spoil timers and more.

2.6.7.5 uint8_t isDST

Is current time DST.

This is non 0 if (current) secTime308Loc() is summertime.

See also

setDST()

2.7 Serial communication

2.7.1 Overview

The primary target hardware for weAutSys — the automation controller weAut_01 — can use AVR's UART0 for serial (V.24 / RS232) communication with optional RTS and/or CTS flow control.

Driver software (see module serial I/O) and system initialisation may bind stdin, stdout and stderr to this serial link. So these streams and all their format string controlled formatting and parsing are available. But these are expensive functions (partly using many 100 µs µController time) and not well suited to weAutSys' non preemptive nature. Hence using the (non blocking) UART and stdI/O functions as well as the formatters and parsers provided here is strongly recommended (en lieu de stdio.h).

As best fit for weAutSys' non preemptive Protothreads approach status functions are provided. They can be used in boolean expressions for Protothreads' blocking and yielding operations:

```
PT_WAIT_ASYIELD_WHILE(pt, outSpace(stdout) < 120);
printf("Hello, world!\n"); // never use printf embedded
```

or better:

```
INFLASH(char hellWo[]) = "Hello, world! \n";
//                        0123456789.1234 5
 : : : : : : :
PT_YIELD_OUT_SPACE(pt, 15);
stdPutS_P(hellWo);
```

Files

- file uart0.h

  Serial communication (basics)

Defines

- #define UART_IN_BUF_CAP 127

  The maximum capacity of the serial input buffer.
- #define UART_IN_SPACE_LIM 11

  UART space limit for flow control.
- #define UART_OUT_BUF_CAP 255

  The capacity of the serial output buffer.

Functions

- uint16_t serInBufferd (FILE ∗streams)

  The number of characters buffered from serial input.
- void setUARTflowcontrolByChar (char cC)

  Sets UART flow control by (one) character.
- uint8_t uartClearInBuffer (void)

  Clear the internal buffer for serial input.
- int uartGetChar (FILE ∗stream)

  Get one byte from serial input.
- uint8_t uartGetLine (char ∗line, uint8_t max)

  Read a line from UART.
- uint8_t uartInBufferd (void)

  The number of characters buffered from serial input.

- uint8_t uartInErrors (void)

  The accumulated serial input (UART0) errors.
- uint8_t uartInRetBufferd (void)

  The number of line feeds buffered from serial input.
- uint8_t uartOutSpace (void)

  The buffer space available for serial output.
- uint8_t uartPutBytes (uint8_t ∗src, uint8_t n)

  Put some bytes (characters) to serial output.
- uint8_t uartPutChars (char ∗src, uint8_t n)

  Put some characters to serial output.
- int uartPutCharsP (char const ∗src, uint8_t n)

  Put some characters from program space to serial output.
- uint8_t uartPutSt (char ∗src, const FILE ∗stream)

  Put a RAM string (some characters) to serial output.
- int uartPutSt_P (prog_char ∗src, const FILE ∗stream)

  Put a string (some characters) from flash memory to serial output.

Variables

- uint8_t uartInAccErrors

  The serial input's accumulated (or) errors.
- uint8_t uartInBufRetCnt

  The serial input buffer line feeds code counter.

2.7.2 Define Documentation

2.7.2.1 #define **UART_OUT_BUF_CAP** 255

The capacity of the serial output buffer.

The value must be 2∗∗n-1 and fit into an unsigned byte.

Possible values are 15, 31, 63, 127 and 255.

Recommended (default) is 255. This is a good compromise concerning RAM saving and avoiding thread waits / yields due to buffer fill. Use lower values only if the RAM is urgently needed.

The capacity's upper limit (255) is fixed by the type (uint8_t) chosen for lengths and indices. This makes most respective operations more than twice as fast as would be with 16 bit types (uint16_t). If larger UART buffers should become necessary that could be changed on customer request. As an alternative using higher baud rates should be considered. At (default) 38400 it takes 67 ms to empty (send) a 256 byte buffer. That is reduced to 22ms with 115K2. Sending more than 100 characters over the UART (every time) in the 10 ms thread (or 23 at 38400) would fail anyway in the long run, no matter the buffer size. Consider Ethernet communication (probably utilising the Telnet) as alternative.

See also

UART_IN_BUF_CAP

2.7.2.2 #define **UART_IN_BUF_CAP** 127

The maximum capacity of the serial input buffer.

The value must be 2∗∗n-1 and fit into an unsigned byte.

Possible values are 15, 31, 63, 127 and 255.

Recommended (default) is 127 that is fit for terminal command lines.

It may be set to to 255 if RAM is available. For still larger buffer sizes compare the remarks about the output buffer capacity. If input size is critical input flow control is the recommended means to put the buffer burden to the sender.

See also

UART_OUT_BUF_CAP

Examples:

main.c.

2.7.2.3 #define UART_IN_SPACE_LIM 11

UART space limit for flow control.

See also

B1D6MODE_CTS_FLOWC

Examples:

main.c.

2.7.3 Function Documentation

2.7.3.1 uint8_t uartOutSpace (void)

The buffer space available for serial output.

This function returns the number of bytes that can be output UART output by serPutChar() e.g. without loosing data.

The returned value is in the range 0 ..UART_OUT_BUF_CAP - 1.

If 0 is returned nothing should be output.

2.7.3.2 uint8_t uartPutChars (char ∗ *src*, uint8_t *n*)

Put some characters to serial output.

This function never blocks. It returns the number of characters transferred. If not enough space is available the returned value will be smaller than `n` or than the length of the string `src`. It may even be 0. In the latter case nothing was output.

Hence this operation should be (indirectly) guarded by uartOutSpace().

The output stops after n characters or at the first 0 in src. By that this is a string output function. To send a (binary) 0 use serPutChar(char c, ...) or uartPutBytes().

Parameters

src	the characters to be output (not NULL !)
n	the maximum number of characters to be output

Returns

the number of characters output

2.7.3.3 uint8_t **uartPutBytes** (uint8_t ∗ *src*, uint8_t *n*)

Put some bytes (characters) to serial output.

This function never blocks. It returns the number of bytes transferred. If not enough space is available the returned value will be smaller than `n`. It may even be 0. In the latter case nothing was output.

Hence this operation should be (indirectly) guarded by uartOutSpace().

Parameters

src	the characters to be output
n	the maximum number of characters to be output

Returns

the number of characters output

2.7.3.4 uint8_t **uartPutSt** (char ∗ *src*, const FILE ∗ *stream*)

Put a RAM string (some characters) to serial output.

This function never blocks. It returns the number of characters transferred. If not enough space is available the returned value will be smaller the length of the string `src`. It may even be 0. In the latter case nothing was output.

Hence this operation should be (indirectly) guarded by uartOutSpace().

The output stops after n characters or at the first 0 in src. By that this is a string output function. To send a (binary) 0 use serPutChar(char c, ...).

Hint: The parameter stream is not used in the present (June 2012) implementation, but should be set correctly in case of future changes.

Parameters

src	the characters to be output
stream	the serial output as stream

Returns

the number of characters output

2.7.3.5 int **uartPutSt_P** (prog_char ∗ *src*, const FILE ∗ *stream*)

Put a string (some characters) from flash memory to serial output.

This function never blocks. It returns the number of characters transferred. If not enough space is available the returned value will be smaller the length of the string `src`. It may even be 0. In the latter case nothing was output.

Hence this operation should be (indirectly) guarded by uartOutSpace().

The output stops after n characters or at the first 0 in src. By that this is a string output function. To send a (binary) 0 use serPutChar(char c, ...).

Hint: The parameter stream is not used in the present (June 2012) implementation, but should be set correctly in case of future changes.

Parameters

src	the characters to be output
stream	the serial output as stream

Returns

the number of characters output

2.7.3.6 int uartPutCharsP (char const * *src*, uint8_t *n*)

Put some characters from program space to serial output.

This function never blocks. It returns the number of characters transferred. If not enough space is available the returned value will be smaller than `n` or the length of the string `src`. It may even be 0. In the latter case nothing was output.

Hence this operation should be (indirectly) guarded by uartOutSpace().

The output stops after n characters or at the first 0 in src.

Parameters

src	the characters to be output in flash memory (not NULL !)
n	the maximum number of characters to be output

Returns

the number of characters output

Examples:

main.c.

2.7.3.7 int uartGetChar (FILE * *stream*)

Get one byte from serial input.

This function never blocks. It returns a byte value read or EOF.

Hence this operation should be (indirectly) guarded by uartInBufferd().

Parameters

stream	(passed by stdio input) totally ignored by this function

Returns

a character 0..255 or EOF

2.7.3.8 uint8_t uartInBufferd (void)

The number of characters buffered from serial input.

This function returns the number of bytes that can be fetched from serial input by uartGetChar(FILE*) without getting EOF.

The returned value is in the range 0 .. UART_IN_BUF_CAP - 1. If 0 is returned nothing should be read. If UART_I-N_BUF_CAP is returned the internal buffer was overrun and the input data are incomplete respectively corrupt.

Examples:

main.c.

2.7.3.9 uint16_t serInBufferd (FILE ∗ *streams*)

The number of characters buffered from serial input.

Same as uartInBufferd (other respectively common signature).

2.7.3.10 uint8_t uartInRetBufferd (void)

The number of line feeds buffered from serial input.

This function gives the number of line feeds, i.e. `LF` or `CR` chars that were read from serial input and not yet fetched from buffer by uartGetChar(FILE∗) or uartGetLine(char[ ], int).

The returned value is in the range 0 .. UART_IN_BUF_CAP . If 0 is returned no (stdio) function expecting a complete line input should be called.

Examples:

main.c.

2.7.3.11 uint8_t uartInErrors (void)

The accumulated serial input (UART0) errors.

This function returns all UART 0 input errors accumulated (ORed) in the past:

Bit 6: buffer overrun. More than UART_IN_BUF_CAP bytes were received but not fetched in due time.

Bit 4: framing error. This usually happens if the two UARTs connected do not have the same setting, one of their clocks are out of tolerance, hardware noise or connector troubles.

Bit 3: UART buffer overrun. This should not happen, the effect of slow input processing is Bit 6. If this happens, some interrupt handlers or (atomic) critical sections block interrupts too long.

Bit 2: Parity error. This has often the same causes as bit 4. Can only happen with parity enabled.

If this function returns 0 all was well since last call or last buffer clearance. A call to this function resets the errors.

2.7.3.12 uint8_t uartClearInBuffer (void)

Clear the internal buffer for serial input.

This function returns the number of characters forgotten. If 0 is returned the buffer was empty anyway.

This function is to be used to recover from buffer overrun or similar errors or to "forget the past" out of any other reasons.

2.7.3.13 uint8_t uartGetLine (char ∗ *line,* uint8_t *max*)

Read a line from UART.

This function will copy all characters to the array (`line`) supplied ending with line feed. This line feed may be one or two characters LF or CR; it will be put as one ending 0 as last character into line.

Returned is the number of characters copied to the array `line`. This might be 0; it will never be more than `max`, uartInRetBufferd() and UART_IN_BUF_CAP.

A full line was got if the last character ([returned value -1]) is (char) 0 which will, as said, replace the terminating line feed form the (buffered) input. So, space permitting, the modified `line`[] will be a null-terminated (C-) string.

To guarantee a full line, a call to this function should be guarded by uartInRetBufferd() and max better be >= UART_IN_BUF_CAP.

Parameters

line	the character array to put the received line to
max	the maximum number of characters transferred

Returns

the number of characters transferred

See also

uartGetCmdLine

2.7.3.14 void setUARTflowcontrolByChar (char *cC*)

Sets UART flow control by (one) character.

i c: input flow control by CTS

o r: output flow control by RTS

b : both CTS and RTS n, all else: no flow control

Parameters

cC	the controlling character (may also be in upper case)

See also

setB1D6mode

2.7.4 Variable Documentation

2.7.4.1 uint8_t uartInAccErrors

The serial input's accumulated (or) errors.

If 0 no errors had occurred since last check.

Bit 4 set: at least one framing error had occurred.

Bit 3 set: at least one overrun error had occured.

Bit 2 set: at least one parity error had occurred.

Bit 6 set: at least one byte could not be put to buffer.

Bit 5 set: at least one spike was ignored.

2.8 LAN / Ethernet communication

2.8.1 Overview

weAutSys provides Ethernet / LAN communication based on

- a ENC28J60 interface (hardware),
- an event driven 28J60 driver and
- uIP [by Adam Dunkels] as TCP/IP stack.

These are the basic functions for all application (support) layers weAutSys supports. There upon application software may implement further protocols and networking applications.

Files

- file network.h

  weAutSys' (low level) system calls, services and types for LAN / Ethernet communication

Data Structures

- struct ipConf_t

  The IP configuration.

Defines

- #define BUF2EH

  The uip buffer cast to Ethernet header.
- #define DHCP_MSK

  Mask for DHCP (use / set by) bit.
- #define DNS1_MSK

  Mask for DNS server 1 set bit.
- #define DNS2_MSK

  Mask for DNS server 2 set bit.
- #define DNS_CLIENT_MSK

  Mask for be DNS client bit.
- #define DNS_MSK

  Mask for DNS servers (set / use if set) bits.
- #define ECHO_PORT 7

  The well known Echo port.
- #define ENC_HWP

  Mask for the ENC HW problem bit in networkNotReady.
- #define ENC_MSK

  Mask for the ENC bits in networkNotReady.
- #define ENC_NRD

  Mask for the ENC not ready bit in networkNotReady.
- #define ENC_PWS

  Mask for the ENC powersave bit in networkNotReady.
- #define IP_ADD(addr0, addr1, addr2, addr3)

  Initializer of an IP (V4) address from four byte values.
- #define IPBUF

The uip buffer cast to Ethernet IP header.

- #define LNC_IDN

  Mask for the link is down (at last status request) in networkNotReady.
- #define LNC_WDN

  Mask for the link was down (before last status request) in networkNotReady.
- #define MODBUS_PORT 502

  The well known Modbus port.
- #define NTP1_MSK

  Mask for NTP server 1 set bit.
- #define NTP2_MSK

  Mask for NTP server 2 set bit.
- #define NTP_CLIENT_MSK

  Mask for be NTP client bit.
- #define NTP_MSK

  Mask for NTP server bits.
- #define NTP_SERVERT_MSK

  Mask for be NTP server bit.
- #define STC_MSK

  Mask for the stack bits in networkNotReady.
- #define STC_NST

  Mask for the stack' IP not set bit in networkNotReady.
- #define TELNET_PORT 23

  The well known Telnet port.

Typedefs

- typedef struct uip_eth_addr eth_addr_t

  Representation of a 48-bit Ethernet address / MAC address.

Functions

- uint16_t deviceDriverPoll (void)

  Hand over received packages.
- void deviceDriverSend (void)

  Send the actual uIP buffer.
- struct uip_eth_addr ∗ getMACforIPaddr (uip_ipaddr_t ipAddr)

  Get the MAC address for an IP address.
- uint8_t linkState (void)

  Check the link state.
- void networkPolling (void)

  Receive and then perhaps send the actual uIP buffer.
- ptfnct_t outEncLanInfoThreadF (hierThr_data_t ∗threadD, FILE ∗toStream)

  Output Ethernet and driver (ENC28J60) state info, the thread function.
- void setMACadd (eth_addr_t ∗mac)

  Set the IP stack's MAC address.
- void setMACaddP (const eth_addr_t ∗mac)

  Set the stack's MAC address.
- void udpAppcall (void)

  The uIP udp event function for the application software.
- void uipAppcall (void)

The uIP event function for the application software.

- void uipGetAddresses (ipConf_t ∗ipConf)

  Get the addresses from the uIP stack.
- void uipSetAddresses (ipConf_t ∗ipConf)

  Set the addresses in the uIP stack.

Variables

- uint8_t arpTickPeriod

  The period of the ARP timeout tick in seconds.
- uint8_t arpTimeOut

  The ARP timeout counter in seconds.
- ipConf_t curIpConf

  The actual IP configuration.
- uint8_t networkNotReady

  Network not ready.

2.8.2 Define Documentation

2.8.2.1 #define IP_ADD(*addr0*, *addr1*, *addr2*, *addr3*)

Initializer of an IP (V4) address from four byte values.

This function constructs an {embraced} initialiser for an uip_ipaddr_t (array) type.
Example:

```
uip_ipaddr_t defaultIPaddr = IPADD(192, 168, 89, 31);
```

will result in the correct array initialisation in network byte order (big endian).

Parameters

addr0	The first octet of the IP address
addr1	The second octet
addr2	The third octet
addr3	The forth octet

Examples:

individEEP.c.

2.8.2.2 #define DHCP_MSK

Mask for DHCP (use / set by) bit.

See also

ipConf_t

Examples:

individEEP.c, and main.c.

2.8.2.3 #define DNS1_MSK

Mask for DNS server 1 set bit.

See also

ipConf_t

Examples:

individEEP.c, and main.c.

2.8.2.4 #define DNS2_MSK

Mask for DNS server 2 set bit.

See also

ipConf_t

2.8.2.5 #define DNS_MSK

Mask for DNS servers (set / use if set) bits.

See also

ipConf_t

2.8.2.6 #define DNS_CLIENT_MSK

Mask for be DNS client bit.

See also

ipConf_t

2.8.2.7 #define NTP1_MSK

Mask for NTP server 1 set bit.

See also

ipConf_t

2.8.2.8 #define NTP2_MSK

Mask for NTP server 2 set bit.

See also

ipConf_t

2.8.2.9 #define NTP_MSK

Mask for NTP server bits.

See also

ipConf_t

2.8.2.10 #define NTP_CLIENT_MSK

Mask for be NTP client bit.

See also

ipConf_t

Examples:

main.c.

2.8.2.11 #define NTP_SERVERT_MSK

Mask for be NTP server bit.

See also

ipConf_t

2.8.2.12 #define BUF2EH

The uip buffer cast to Ethernet header.

This macro provides the buffer uip_buf as type uip_eth_hdr .

2.8.2.13 #define IPBUF

The uip buffer cast to Ethernet IP header.

This macro provides the buffer uip_buf as type ethip_hdr .

2.8.3 Typedef Documentation

2.8.3.1 typedef struct uip_eth_addr eth_addr_t

Representation of a 48-bit Ethernet address / MAC address.

This is uIP's type uip_eth_addr defined in uip.h holding an array uint8_t addr[6] in the end.

addr[0] would be octet [1] of the MAC address defining

Bit 0 = 0: unicast / = 1: multi/broadcast address

Bit 1 = 0: manufacturer / =1: locally assigned address

2.8.4 Function Documentation

2.8.4.1 void setMACadd (eth_addr_t ∗ *mac*)

Set the IP stack's MAC address.

This sets the the Ethernet stack's i.e. uIP's MAC address but not the driver's (ENC28J60).

Of course, the driver's MAC setting has to be same. This is done by encSetMacAdd(), encInit() or networkInit().

See also

uip_setethaddr
actMACadd
encSetMacAdd

Parameters

mac	pointer to a structure (in RAM) holding the new MAC address

2.8.4.2 void setMACaddP (const eth_addr_t ∗ *mac*)

Set the stack's MAC address.

This is the same as setMACadd except for the new MAC address structure held in program (flash) memory

Parameters

mac	pointer to a structure (in flash) holding the new MAC address

2.8.4.3 void uipSetAddresses (ipConf_t ∗ *ipConf*)

Set the addresses in the uIP stack.

This function takes the fields ipAddr, netMask, defaultRouter and dnsAddr and makes the appropriate settings in the uIP stack. The bit 4 ("stack's IP is not yet set") is cleared in networkNotReady.

`ipConf's` cFlags is irrelevant: the settings are made even if the flag says "use DHCP". Hence `ipConf` must not be null and it must have correct entries for `ipAddr`, `netMask` and `defaultRouter`.

See also

uip_sethostaddr
uip_setnetmask
uip_setdraddr
resolv_conf

Parameters

ipConf	pointer to the source of the addresses and masks

2.8.4.4 void uipGetAddresses (ipConf_t ∗ *ipConf*)

Get the addresses from the uIP stack.

This function fills the fields ipAddr, netMask, defaultRouter and dnsAddr from the respective settings in the uIP stack.

ipConf's cFlags is left unchanged.

See also

uipSetAddresses

Parameters

ipConf	pointer to the destination of the addresses and masks

2.8.4.5 void **deviceDriverSend** (void)

Send the actual uIP buffer.

See function devicedriver_send respectively ethernetdevicedriver_send in uIP documentation.

This function does nothing if the NIC's transmit logic is not (yet) idle. In case of weAut_01 the NIC is an ENC28J60.

2.8.4.6 void **networkPolling** (void)

Receive and then perhaps send the actual uIP buffer.

This is standard control structure around calling uip_input as described in the uIP documentation in one function.

To have all this in one block by calling this function may be questionable. It will be put in (pieces in) appropriate threads. And as of this writing this function was not used yet.

2.8.4.7 uint16_t **deviceDriverPoll** (void)

Hand over received packages.

This function checks if the network driver IC (NIC), an ENC28J60 in case of weAut_01, has received a package. If so it is copied to uIP's uip_buf and the number of bytes is returned.

See function devicedriver_poll respectively ethernetdevicedrver_poll in uIP documentation.

Returns

the length of the (new) package or 0 if there was none

2.8.4.8 ptfnct_t **outEncLanInfoThreadF** (hierThr_data_t ∗ *threadD*, FILE ∗ *toStream*)

Output Ethernet and driver (ENC28J60) state info, the thread function.

This thread will output some info on Ethernet and driver configuration and state to `toStream` (as standard output).

This thread function acts on threadD's secondary thread and secondary flag. `flag2's` bits signal which informations are to be output:

Bit 0 : MAC

Bit 1 : current IP setting

Bit 2 : default IP setting

Bit 3 : ARP

Bit 4 : Link state

Bit 5 : DHCP state

As secondary thread it will usually be started as the primary threads child.

Parameters

threadD	the thread state data (pointer to; not NULL)
toStream	streams to switch the standard streams to (if not MULL)

Returns

PT_YIELDED if waiting for conditions that may become automatically true; PT_WAITING if blocked by conditions that other's action may set true; PT_EXITED if stopped by conditions that never will become true (sorry, not all work done and never will; PT_ENDED if ready with all work

2.8.4.9 uint8_t **linkState** (void)

Check the link state.

This function checks the actual link state and sets the relevant bits (LNC_IDN and LNC_WDN) in networkNotReady.

Returns

0: link is down; 1: link is and was up since last call; 2: link is up, but was down since last call

2.8.4.10 struct **uip_eth_addr*** **getMACforIPaddr** (**uip_ipaddr_t** *ipAddr*) `[read]`

Get the MAC address for an IP address.

This function checks the ARP table for an entry about ipAddr and returns the (pointer to its) MAC address.

The caller must not manipulate the returned MAC.

Parameters

ipAddr	IP address the MAC is to be determined for

Returns

pointer to MAC if determinable or NULL else

2.8.4.11 void **udpAppcall** (void)

The uIP udp event function for the application software.

The reason for being called will be in the `uip_udp_conn` structure, especially in the ports.

This function will be called for events, ports etc. not handled by system software (like NTP or DHCP e.g.).

If no extra protocols are implemented by user / application software the implementation should do nothing (but `return;`).

Otherwise it is best to use (at least) the remote port to distribute to different protocol handling functions (best being protothreads). Client example:

```
void udpAppcall(void){

   if(uip_udp_conn->rport == HTONS(MTP_PORT)) {  // MTP protocol
     mtpAppcall(); // protothread function / state machine handling MTP
     return;
   }  // MTP protocol

   if(uip_udp_conn->rport == HTONS(THCPC_SERVER_PORT)) {  // THCP protocol
      thcpc_appcall(); // protothread function / state machine handling THCP
      return;
   } // THCP protocol (client)

   // add further protocols here

} //  udpAppcall()
```

See also

udp_appcall

Examples:

main.c.

2.8.4.12 void uipAppcall (void)

The uIP event function for the application software.

The reason for being called will be in `uip_flags`.

This function will be called for events, ports etc. not handled by system software.

If no extra protocols are implemented by user / application software the implementation should do nothing (but `return;`).

Otherwise it is best to use (at least) the local port to distribute to different protocol handlers. The handlers would best be implemented as protothread functions using appropriate `appstate` fields in the `uip_conn` as state. The server example is simplified (i.e. stateless):

```
void uipAppcall(void){

   // for state use something
   // like : struct cliThr_data_t *s = (cliThr_data_t *)&(uip_conn->appstate);

   if(uip_conn->lport == HTONS(7)    // Echo protocol .. and
         && (uip_newdata() || uip_acked())) { // new (or ack'ed) data
      uip_send(uip_appdata, uip_datalen());
      return;
   } // echo server (new data only)

   // add further handling and protocols here

} // uipAppcall()
```

See also

uip_appcall

Examples:

main.c.

2.8.5 Variable Documentation

2.8.5.1 ipConf_t curIpConf

The actual IP configuration.

See also

defaultConfData defaultConfData.ipConf

Examples:

main.c.

2.8.5.2 uint8_t arpTickPeriod

The period of the ARP timeout tick in seconds.

The standard / default value is 10. User / application software might set a higher value to reduce ARP µController load.

See also

UIP_ARP_MAXAGE
arpTimeOut

2.8.5.3 uint8_t arpTimeOut

The ARP timeout counter in seconds.

This counter is decremented every second. At 0 it is reset to arpTickPeriod and uip_arp_timer() is called. The latter uIP function handles the aging of ARP entries.

See also

UIP_ARP_MAXAGE
arparpTickPeriod

2.8.5.4 uint8_t networkNotReady

Network not ready.

If this state flag 0x00 means the network is ready.

One of the lower four bits stands for a network state or problem that impedes communication.

The upper four bits do the same for the hardware driver, which in the case of weAut_01 it is an ENC28J60.

Bit 7 : ENC is not initialised yet / was reset (mask ENC_NRD)

Bit 6 : ENC has hardware (CLKRDY) trouble (mask ENC_HWP)

Bit 5 : ENC is in power save (PWRSV) mode (mask ENC_PWS)

Bit 3 : stack's IP is not yet set / determined (by DHCP e.g., mask STC_NST)

Bit 2 : link is down (status of last last request by e.g. linkState(), mask LNC_IDN)

Bit 1 : link was down (status before last last request by e.g. linkState(), mask LNC_WDN)

2.9 Communication with a small memory card (via SPI)

2.9.1 Overview

These are the basic functions to communicate serially with a small memory card. Higher level capabilities as e.g. a file system are build upon those functions.

Files

- file smc.h

  weAutSys' (low level) system calls, services and types for communication with a small memory card

Data Structures

- struct smcThr_data_t

  The organisational data for a small memory card (SMC) handling thread.

Defines

- #define APP_CMD 55

  SMC command: next command is application specific.
- #define GO_IDLE_STATE 0

  SMC command: soft reset.
- #define LOCK_UNLOCK 42

  SMC command: lock / unlock the card.
- #define NCR_EXTR_WAIT 8

  Maximum extra waits for command response.
- #define othersAskPrio()

  Other devices asks for priority.
- #define READ_OCR 58

  SMC command: read the card's OCR register (R3)
- #define READ_SINGLE_BLOCK 17

  SMC command: read one block.
- #define SEND_CID

  SMC command: get CID.
- #define SEND_CID

  SMC command: get CID.
- #define SEND_CSD 9

  SMC command: send card specific data.
- #define SEND_IF_COND 8

  SMC command: send interface condition.
- #define SEND_OP_COND 1

  SMC command: init. process.
- #define SEND_OP_COND_APP

  SMC application specific command: initialisation process.
- #define SEND_STATUS 13

  SMC command: get status (R2)
- #define SET_BLOCKLEN 16

  SMC command: set block length for LOCK_UNLOCK.
- #define smcInsPow()

Inserted card is powered up.

- #define smcReceive()

  Receive a single byte from the small memory card.
- #define smcReceiveN(receiveB, skip, n)

  Receive n bytes from the SMC to a buffer with optional skip.
- #define smcTypeD()

  The SMC's type is determined and OCR is known.
- #define smcXmit(datByte)

  Transmit a single byte to the small memory card.
- #define smcXmit2(sendB1, sendB2)

  Send two bytes and receive one byte to/from the small memory card.
- #define WRITE_BLOCK 24

  SMC command: write one block.

Functions

- uint8_t checkBusy (uint8_t tries)

  Check if card is busy.
- void clk80 (void)

  Have 80 dummy SPI clocks.
- uint8_t crc7stp (uint8_t crcIn, uint8_t datByte) __attribute__((always_inline))

  Calculate CRC7 (one step)
- void deSelectSMC (void) __attribute__((always_inline))

  De-select the small memory card.
- uint8_t doSectorRead (uint32_t sector)

  Order sector read (as background task)
- uint8_t doSectorSync (void)

  Order sector synchronisation (as background task)
- uint8_t getSMCtype (void)

  Determine the card type.
- void initSMCthreadState (uint8_t actionFlag)

  Initialise the small memory card (smc) handling thread.
- uint8_t readDataBlock (uint32_t sector, uint8_t ∗buff)

  Read single data block (512 byte)
- uint8_t sendAppCmd (uint8_t appCmdNum, uint32_t cmdArg)

  Send an application specific command to the small memory card.
- uint8_t sendCmd (uint8_t cmdNum, uint32_t cmdArg)

  Send a command to the small memory card.
- uint8_t sendCmd0arg (uint8_t cmdNum)

  Send a command with 0 argument to the small memory card.
- void setSectorModified (uint32_t sector)

  Mark sector buffer data as modified (start)
- uint32_t smcGetSectCount (void)

  Get the sector count.
- uint8_t smcInsertSwitch (void)

  Hardware card detect.
- uint8_t smcReadCID (uint8_t ∗buff)

  Read the SMC's CID.
- uint8_t smcReadCSD (void)

  Read the SMC's CSD.

- uint8_t smcReadOCR (void)

    Read the SMC's ORC.
- void smcSetFrq (uint8_t frqUse) __attribute__((always_inline))

    Set the SPI clock frequency for the small memory card.
- uint8_t smcSetIdle (uint8_t mode)

    Put card to idle state.
- ptfnct_t smcThreadF (void)

    The small memory card (smc) handling thread.
- uint8_t writeDataBlock (uint32_t sector, const uint8_t ∗buff)

    Write single data block (512 byte)

Variables

- struct smcThr_data_t smcState

    The (one) small memory card (SMC) state.

2.9.2 Define Documentation

2.9.2.1 #define **othersAskPrio( )**

Other devices asks for priority.

This expression is true if other software respectively devices need resources used by SMC operations in critical situations.

As some SMC operations need SPI communication or thread time uninterrupted for quite a long time they should refrain from doing so as long as this request is pending.

For SMC processing controlled by this module's functions this time may be up to 2ms as documented in each case. File system implementations implemented upon these (driver) functions would increase this value quite considerably if programmed unaware of the (Protothread) threading model.

As of Revision 422++ this request comes from (seldom) NTP synchronisation actions only.

2.9.2.2 #define **NCR_EXTR_WAIT** 8

Maximum extra waits for command response.

The maximum number of (8 SPI clock) cycles to receive a valid command response is this value + 1. The specification says the number (NCR) of those waiting cycles is in the range of 1 .. 8 (depending on card type and command). Hence 7 would be the minimum for NCR_EXTR_WAIT; anything larger than 10 would just waste time in case of a (command) failure.

2.9.2.3 #define **SEND_CID**

SMC command: get CID.

SMC command: send card idendification.

2.9.2.4 #define **SEND_CID**

SMC command: get CID.

SMC command: send card idendification.

2.9.2.5 #define LOCK_UNLOCK 42

SMC command: lock / unlock the card.

See also

SET_BLOCKLEN

2.9.2.6 #define SEND_OP_COND_APP

SMC application specific command: initialisation process.

Like SEND_OP_COND this command activates the card's initialisation process. But as an application specific command it has to be prepended by APP_CMD() or simply used in sendAppCmd instead of sendCmd().

Note: In SPI mode SEND_OP_COND_APP and SEND_OP_COND should have the same behaviour, though most implementors prefer the application specific variant. This tradition is respected here.

2.9.2.7 #define smcXmit(*datByte*)

Transmit a single byte to the small memory card.

Parameters

datByte	the byte to be sent to the SMC

Returns

the byte received (on MiSo); can be disregarded for SMCs (0 may indicate an error)

2.9.2.8 #define smcXmit2(*sendB1*, *sendB2*)

Send two bytes and receive one byte to/from the small memory card.

Parameters

sendB1	the byte to be sent first
sendB2	the second byte to be sent

Returns

the byte read from SPI while transmitting; can be disregarded for SMCs `sendB2` (0 may indicate an error)

2.9.2.9 #define smcReceive( )

Receive a single byte from the small memory card.

Returns

the byte received

See also

smcReceiveN

2.9.2.10 #define smcReceiveN(*receiveB,* *skip,* *n*)

Receive n bytes from the SMC to a buffer with optional skip.

This function does `skip` + `n` receptions. The (last) `n` bytes received are put into the buffer `receiveB`. It returns nothing (void).

Compared to doing this by `skip` + `n` times smcReceive() this function is at least 20 % faster.

Parameters

receiveB	pointer to the buffer (type uint8_t ∗) to receive `n` bytes to (must not be NULL if `n` != 0)
skip	if $>$ 0 `skip` bytes are received and forgotten before `receiveB` will be filled
n	number of bytes to be received and filled into `receiveB`

2.9.2.11 #define smcInsPow( )

Inserted card is powered up.

This expression is true if a SMC is inserted and powered up according to smcState.

2.9.2.12 #define smcTypeD( )

The SMC's type is determined and OCR is known.

This expression is true if a (inserted and powered up) SMC's type has been determined. In the process also the OCR has been read to smcState.ocr.

2.9.3 Function Documentation

2.9.3.1 uint8_t smcInsertSwitch (void)

Hardware card detect.

On the weAut_01 module the card switch must be "jumpered" to port D6 for this function to work properly.

Note

To see a card that was correctly powered up and put out of idle state has vanished is easily done by every command. So in this respect the card switch is hardly needed. On the other hand the detection of a potentially inserted SMC by pure (SPI) access requires the complete power up and type recognition process.

Remark: It might well be considered a misconception in weAut_01 not to tie the respective port (D5) directly to the card insert switch and have a lot of jumper options instead. This can be mended by an (1K) resistor soldered in 2 via holes near the jumper.

Returns

true as 'i': switch closed = card inserted, true as 'm': may be inserted, switch not configured; false (0): no card inserted switch open.

2.9.3.2 void clk80 (void)

Have 80 dummy SPI clocks.

Some small memory cards need at least 74 (dummy) SPI clocks with their chip select inactive (high) and data in (DI, MoSi) all ones after power up.

This function sends 80 clocks (sClk) on the SPI (2) interface used for small memory cards with all (SPI2) devices de-selected. The stream of 80 clocks delivered by this function has no gaps.

At SPI clock frequency of 2 MHz the whole thing would take 40,4 µs. At the 400 kHz usually required for power up it will take 200 µs.

See also

smcSetIdle()

2.9.3.3 void **smcSetFrq** (uint8_t *frqUse*)

Set the SPI clock frequency for the small memory card.

Some (older) small memory cards require SPI clock frequencies of 400 kHz (or lower) during initialisation. Afterwards higher SPI baud-rates are to be used. This function sets the latter / normal mode frequency.

A 20 MHz clocked micro-controller (like ATmega1284P) has a maximum 10 MHz SPI baud rate. There are few SMC types with a maximum SPI clock frequency of 6.5 MHz around. Most other types can handle much higher clock rates. So the principally possible 10 MHz would hardly give them a wet shirt.

The next possible lower frequency is 5 MHz. It would fit all card types, anyway. Additionally in case of resistor divide level shifters — from ATmega's 5V to SMC's 3.3V — the waveforms provided for some card types dictate 5 MHz as maximum SPI clock frequency.

Parameters

frqUse	control value to set SPI 2 clock for the memory card (use the predefined values or the UBRR1 formula)

See also

spi2EtherChipBaud
spi2MemCardBaud

2.9.3.4 uint8_t **crc7stp** (uint8_t *crcIn,* uint8_t *datByte*)

Calculate CRC7 (one step)

This function calculates the CRC7 (used in some small memory cards for command transmission). It is a 7 bit CRC with polynomial x7 + x3 + 1. This function implements just the step for next 8 bits given by `datByte`. The result returned would have to be fed as `crcIn` for the next step. Use 0 for `crcIn` in the first step.

To make the final 7 bit CRC sendable to the small memory card via SPI it must be put in the upper seven bit of the CRC byte padded with an one in bit 0 by:

crc = (crc<<1) | 1; // make so sendable after the last step

The function uses 10 CPU cycles if inlined as intended or 18 including call and return. Hence the algorithm is "faster than one byte" for all SPI clocks not faster than 1/2 CPU clock. Sending one byte at 5 MHz takes 48 CPU clocks with a 20 MHz clocked ATmega.

Parameters

crcIn	the crc before; use 0 in first step
datByte	the next 8 bits for the CRC

Returns

the new CRC

2.9.3.5 void deSelectSMC (void)

De-select the small memory card.

This function (chip-) de-selects the SMC. It has to be used when all commands and write / reads are done. An explicit select is not needed as all command functions do so automatically.

Hint: Most operations in most SMCs (probably all) fail if de-selected and re-selected. Keep in mind that small memory cards are not very friendly to sharing a SPI bus with other devices.

2.9.3.6 uint8_t sendCmd (uint8_t *cmdNum*, uint32_t *cmdArg*)

Send a command to the small memory card.

This function sends the command, the arguments and the CRC7 to the memory card. Returned is the card's answer if any. Several (i.e. NCR_EXTR_WAIT + 1) attempts are made to get the cards command response recognised by bit 7 zero. If the response returned is 0xFF the card may be gone.

In this (worst) case this function takes 74 µs at a SPI clock frequency of 2 MHz compared to 32.5 µs for a card answering fast.

Hint: To send an application specific command use the function sendAppCmd().

Hint2: If the response is 0 (all OK) and if the command's specified answer is not of type R1 (or if it is a block read command), its up to the caller of this or similar functions to receive all coming response bytes, using e.g. smcReceive() or smcReceiveN().

Hint3: The returned response is also stored in smcState.lastCmdResp[0].

Parameters

cmdNum	command number, only the lower 6 bits are relevant
cmdArg	the 4 byte (32 bits) command arguments

Returns

the first (and in case of R1 only) response byte

See also

GO_IDLE_STATE
deselectSCM()
sendCmd0arg
sendAppCmd

2.9.3.7 uint8_t sendCmd0arg (uint8_t *cmdNum*)

Send a command with 0 argument to the small memory card.

This function is equivalent to

sendCmd(cmdNr, 0);

with some improvements for this frequent case.

See also: The hints at sendCmd().

Parameters

cmdNum	command number, only the lower 6 bits are relevant

Returns

the first (and in case of R1 only) response byte

2.9.3.8 uint8_t sendAppCmd (uint8_t *appCmdNum*, uint32_t *cmdArg*)

Send an application specific command to the small memory card.

This function is equivalent to

sendCmd0arg(APP_CMD);

sendCmd(cmdNr, cmdArg);

besides some slight improvements.

See also: The hints at sendCmd() and SEND_OP_COND_APP.

Parameters

appCmdNum	application specific command number (lower 6 bits)
cmdArg	the 4 byte (32 bits) command arguments

Returns

the first (and in case of R1 only) response byte

2.9.3.9 uint8_t smcSetIdle (uint8_t *mode*)

Put card to idle state.

This function should run once a small memory card is inserted (or is believed to have been). Returned is the "R1" answer to the soft reset command. Only the value 1 is OK meaning card is present and (now) in idle state. All other values are faults. 0xFF (no valid R1 response) usually means no card inserted or card not responding at all.

Most specifications require 400 kHz for the initial ref clk80 "dummy clocks" and the command. Though this seems out-dated for most modern cards it can for compatibility reasons be done by setting bit 0 in `mode`. The whole thing takes 370µs then.

In smcState.cardType this function sets bits CT_POWERD_UP and CT_INSERTED on success; all other bits are cleared.

Parameters

mode	Bit 0 set (1): force 400 kHz and restore spi2MemCardBaud to previous value afterwards Bit 1 set (2): omit the 80 dummy clocks before the go idle command Bit 2 set (4): omit the go idle command (0 is returned in that case)

Returns

answer (R1) for the command sent: 0x01 is OK (card in slot and in idle state)

2.9.3.10 uint8_t checkBusy (uint8_t *tries*)

Check if card is busy.

This function returns 0 (false not busy) if the SMC is inserted and responds 0xFF at least twice to two dummy reads. The two consecutive not busy answers are looked for 3 .. (3 + `tries`) times.

If the outcome is not busy the card is not de-selected.

Parameters

tries	2..255 maximum number of dummy check reads (byte clock cycles) used

Returns

0: not busy; 0xFF: no card inserted and powered up; 1: busy

2.9.3.11 ptfnct_t smcThreadF (void)

The small memory card (smc) handling thread.

If some basic handling, like initialising, has to be done with a SMC this protothread thread will do it reasonable steps, dividing long running procedures by yielding.

Hint: As this is for small (petit) systems with just one SMC slot this thread uses the one smcState (without any choice by parameter).

2.9.3.12 void initSMCthreadState (uint8_t *actionFlag*)

Initialise the small memory card (smc) handling thread.

This function resets all SMC state. It should be called on reset / restart and on SMC removal, failures and insertion.

Parameters

actionFlag	start value for the SMC thread's flag

See also

smcState

2.9.3.13 uint8_t smcReadOCR (void)

Read the SMC's ORC.

This function reads the operation condition register (OCR, 4 byte) of an inserted and powered up small memory card. In case of success 0 is returned and the result is stored in smcState.ocr.

It is usually not necessary to call this function if smcTypeD() is true as the OCR and the CSD have been read during the card type determination process.

Returns

0: OK or negative response of command READ_OCR or or 0x80 if SCM isn't powered up.

2.9.3.14 uint8_t smcReadCSD (void)

Read the SMC's CSD.

This function reads the card specific data register (CSD, 16 byte) of an inserted and powered up small memory card. In case of success 0 is returned and the result is stored in smcState.csd.

It is usually not necessary to call this function if smcTypeD() is true as the OCR and CSD have been read during the card type determination process.

Returns

0: OK or negative response of command SEND_CSD or or 0x80 if SCM isn't powered up.

2.9.3.15 uint32_t smcGetSectCount (void)

Get the sector count.

If the card type is determined this function returns the number of (512 byte) sectors on the card and 0 otherwise.

2.9.3.16 uint8_t smcReadCID (uint8_t * *buff*)

Read the SMC's CID.

This function reads the card identification register (CID, 16 byte) of an inserted and powered up small memory card. In case of success 0 is returned and the result is stored in `buff`.

0: Manufacturer ID

1,2: OEM/Application ID

3..7: Product name

8: Product revision

9..12: Serial number

13 15: Manufaturing date

Parameters

buff	16 byte to write the CID to

Returns

0: OK or negative response of command SEND_CSD or or 0x80 if SCM isn't powered up.

2.9.3.17 uint8_t readDataBlock (uint32_t *sector,* uint8_t * *buff*)

Read single data block (512 byte)

This function does one single block respectively sector read — with block size being fixed to 512 byte here. In case of success 0 is returned. If the given `sector` number has already been cached the SMC is not accessed.

In case of failure non 0 is returned. More information can be found in smcState.

In the normal case of SCC access the time needed by this function heavily depends on the SMC make. The (astonishing) differences are mainly due to different waits for the so called data token. The time needed for this With ∼31 waits for data token (512MByte KData card) this takes about 1 ms. With 181 checks for the data token (8GByte Verbatim) we need 1,35 ms but this varies up to 250 waits and about 2 ms.

weAutSys is a non pre-emptive system — and besides that SMC operations (the /CS) could not be interrupted anyway. Hence it is clear and was said at other places this block read operation is the most critical for an automation modul / system with an 1ms cycle. No two such operations shall follow each other without yielding — and multi-block SMC operations are, of course, out of the question.

Hint: This function as well as writeDataBlock() handle the sector to byte address transformation for commands READ_SINGLE_BLOCK resp. WRITE_BLOCK needed with low capacity SMC types internally. Parameters `sector` (and smcState.rwSector) always reflect the sector number (0, 1 ...).

Parameters

sector	the sector number
buff	pointer to where to write the received data (NULL not allowed)

Returns

0: OK or negative response of command READ_SINGLE_BLOCK or or 0x80 if no data token came (time out; may try later) or no no type info available

See also

writeDataBlock

2.9.3.18 uint8_t writeDataBlock (uint32_t *sector,* const uint8_t ∗ *buff*)

Write single data block (512 byte)

Parameters

sector	the sector number
buff	pointer to where to get the send data from (NULL not allowed)

Returns

0: OK or negative response of command WRITE_BLOCK or or 0x80 if no type info available

See also

readDataBlock

2.9.3.19 uint8_t getSMCtype (void)

Determine the card type.

If the an SMC is inserted and its type has been determined this function returns the type bits set in smcState.card-Type.

Otherwise 0 is returned and the card type determination is triggered in the SMC handling thread.

Returns

0: no type determined yet, CT_V_1, CT_V_2 [| CT_HC], CT_V_3,

2.9.3.20 uint8_t doSectorRead (uint32_t *sector*)

Order sector read (as background task)

This function sets smcState.rwSector by `sector` and initialises smcState.sectorBuff to be read from that sector.

Parameters

sector	the (new) sector number for smcState.rwSector/.sectorBuff

Returns

0: OK already done (smcState.sectorBuff in sync.),
1: will be done ASAP (in background task), call again later,
0x80: action not possible (no card, no type or other error)

See also

setSectorModified

2.9.3.21 void **setSectorModified** (uint32_t *sector*)

Mark sector buffer data as modified (start)

This function sets smcState.rwSector by `sector` and marks as smcState.sectorBuff as modified (new) data for the respective sector. This neither performs any read or write operations nor orders such as background task.

Hint: If this functions changes the current buffer's (smcState.sectorBuff) sector number and the buffer contained modified data for that previous sector those modifications will be lost. Call doSectorSync() before if that would not be intended.

Parameters

sector	the (new) sector number for smcState.rwSector/.sectorBuff

See also

doSectorRead

2.9.3.22 uint8_t **doSectorSync** (void)

Order sector synchronisation (as background task)

This function checks if smcState.sectorBuff contains modified data (for sector smcState.rwSector) and if so initialises smcState.sectorBuff to be written to that sector.

Example sector copy:

```
PT_WAIT_ASYIELD_WHILE(&myThread, doSectorRead(sourceSect) == 1);
if (doSectorSync()) {  optional error handling / exit  }
setSectorModified(destinationSect);
PT_WAIT_ASYIELD_WHILE(&myThread, doSectorSync() == 1);
if (doSectorSync()) {  optional error handling / exit  }
```

Returns

0: OK already done (smcState.sectorBuff in sync.),
1: will be done ASAP (in background task), call again later,
0x80: action not possible (no card, no type or other error)

2.10 + + Application (layer) support + +

2.10.1 Overview

weAutSys brings some support for all automation / user / application programming and for network applications. The latter supplement those already brought by Adam Dunkels' TCP/IP stack uIP.

At the upper stack levels uIP brings some protocol and application support.

weAutSys adapted and uses ARP, DNS (resolve) and the DHCP client and implements its own NTP client and a Telnet and a Modbus server".

Modules

- Common types and helpers
- Command line interpreter (CLI)
- NTP client
- Telnet server
- Modbus server
- Streams
- Files

2.11 Common types and helpers

2.11.1 Overview

Some weAutSys of types and helper macros and functions have common usages in more than one (application) module.

Clearly, in this category fall definitions due to special treatment sometimes required by parts of the software toolchain (Eclipse, SVN, GCC, Doxygen, AVR...). To give just two examples:

Doxygen gets problems with some AVR / Atmel related attributes (that clearly fall outside any C standard).

Some Eclipse versions have problems to follow the quite complicated macro expansion and include file trees controlled by the µController type in the AVR GCC lib.

See also the text blocks and utilities.

Files

- file common.h

  weAutSys' common types and helper functions
- file config.h

  weAutSys' configuration settings
- file ll_common.h

  weAutSys' basic common types and helper functions

Data Structures

- struct cliThr_data_t

  The organisational data for a command line interpreter (CLI) thread.
- struct hierThr_data_t

  The organisational data for a small thread hierarchy.
- struct mThr_data_t

  State data for a thread (minimal)
- struct outFlashTextThr_data_t

  The organisational data for a flash strings array output thread.
- struct thr_data_t

  State data for a thread (universal variable type)
- struct u16div_t

  Two unsigned words intended for quotient and remainder.
- struct u8div_t

  Two unsigned bytes intended for quotient and remainder.
- union ucnt16_t

  A medium (16 bit) value in different resolutions.
- union ucnt32_t

  A big (32 bit) value in different resolutions.

Defines

- #define ADDR_T

  flash address is 16 bit for this µController
- #define ADDR_U

  flash address is 16 bit for this µController

- #define addr_v

  flash address is 16 bit for this µController
- #define APP_END

  The end of the application flash area (as byte address + 1)
- #define BEL

  The bell code.
- #define BOOTL_BEG

  The start address of the bootloader (area)
- #define BOOTSIZE 4096

  The (maximum) boot size in bytes.
- #define BS

  The back space code.
- #define CR

  The carriage return code.
- #define ESC

  The Escape code.
- #define EXTRA_THR_ST_SZ 158

  Size of (extra) thread state data.
- #define FCPU_S

  The CPU clock frequency (as string)
- #define FF

  The form feed code.
- #define FOLLOW_UP

  Follow up string marker for outFlashTextThr_data_t.
- #define GN_LED_INV

  invert gn LED: ! ; no invert: empty
- #define HBYTE(x)

  Get the high byte (of a 16 bit value)
- #define HT

  The horizontal tab code.
- #define LARGE_MEMORY 0

  The flash memory byte address type.
- #define LBYTE(x)

  Get the low byte (of a 16 bit value)
- #define LEN_OF_CLITHR_LINE

  The maximum number of characters in cliThr_data_t.line.
- #define LF

  The linefeed code.
- #define OFF

  A boolean false respectively off.
- #define ON

  A boolean true respectively on.
- #define PLATFORM_S

  The platform name (as string)
- #define RD_LED_INV

  invert rd LED: ! ; no invert: empty
- #define SIZE_OF_BIGGEST_APPSTATE 164

  The size of the biggest application state structure used in bytes.
- #define STD_BAUD 38400

  Standard resp. used baudrate.
- #define SYST_AUT "Albrecht Weinert <a-weinert.de> "

The runtime's author.

- #define SYST_BLD

  The runtime's system build and time.
- #define SYST_COP

  The runtime's copyright notice.
- #define SYST_DAT "$Date: 2015-08-04 15:16:04 +0200 (Di, 04 Aug 2015) $"

  This file's last modification date (SVN)
- #define SYST_MOD

  This file's last modifier respectively SVN author.
- #define SYST_NAM "weAutSys"

  The runtime's name.
- #define SYST_REV

  The runtime's revision.
- #define TRAMPOLIN_USE 1

  The trampolin hack is used.
- #define USART_IN_APP_WITH_INT

  UART uses no interrupt in application.
- #define VT

  The vertical tab code.

Defines due to tools

- #define INFLASH(decl)

  Alternative declaration for flash memory.
- #define INEEPROM(decl)

  Declaration for EEPROM memory.
- #define STR(leMac)

  A macro value as string.

Typedefs

- typedef ptfnct_t(fun_t)(struct thr_data_t ∗thrData)

  Type of a protothread function (struct thr_data_t ∗)
- typedef ptfnct_t(funA_t)(struct cliThr_data_t ∗thrData)

  Type of a protothread function (cliThr_data_t ∗)
- typedef ptfnct_t(funU_t)(struct mThr_data_t ∗uthr_data)

  Type of a protothread function (struct mThr_data_t ∗)
- typedef ptfnct_t(funV_t)(void)

  Type of a protothread function (void)
- typedef funU_t ∗ p2ptFun

  Pointer to a protothread function (struct mThr_data_t ∗)
- typedef funA_t ∗ p2ptFunA

  Pointer to a protothread function (cliThr_data_t ∗)
- typedef fun_t ∗ p2ptFunC

  Pointer to a protothread function (struct thr_data_t ∗)
- typedef funV_t ∗ p2ptFunV

  Pointer to a protothread function (void)

Variables

- char const bLF1 [ ]

  1 blank, 1 linefeed 0 terminated in flash memory
- char const bLF2 [ ]

  1 blank, 2 linefeed 0 terminated in flash memory
- char const l4nefeeds [ ]

  A short string with just one blank and four linefeeds in flash memory.

2.11.2 Define Documentation

2.11.2.1 #define SIZE_OF_BIGGEST_APPSTATE 164

The size of the biggest application state structure used in bytes.

The type of the application state that is to be stored in the uip_conn structure may vary. This number shall be the maximum size of all application states.

Note

This would have to be about 2 Kbyte as soon as A.D.'s HTTP implementation is used. This (one of the weakest parts of uIP) will can not be handled this way for small embedded systems.

See also

uip_conn

2.11.2.2 #define FOLLOW_UP

Follow up string marker for outFlashTextThr_data_t.

See also

outFlashTextThr_data_t

Examples:

main.c.

2.11.2.3 #define SYST_NAM "weAutSys"

The runtime's name.

See also: SYST_DAT

2.11.2.4 #define SYST_DAT "$Date: 2015-08-04 15:16:04 +0200 (Di, 04 Aug 2015) $"

This file's last modification date (SVN)

This file's subversion (SVN) modification date and revision number are taken as the weAutSys runtime's revision data.

The value is a "quoted" string.

See also: DEFAULT_START_TIME SYST_REV SYST_MOD

2.11.2.5 #define SYST_MOD

This file's last modifier respectively SVN author.

The value is a "quoted" string.

See also: DEFAULT_START_TIME SYST_REV SYST_DAT

2.11.2.6 #define TRAMPOLIN_USE 1

The trampolin hack is used.

This value is true (!= 0) if the trampolin hack (+ linker relaxation) is used (and hopefully works) on large ATmegas with more than 128K flash.

As of January 2014 this is a "feature" of arv-gcc, thus avoiding the introduction of 24 bit pointers (for label as value) as well as a correct and consistent handling of EIND.

This marco defines a property of the (AVR gcc) toolchain. The only place it is used (or prepared to be used as of Nov. 2014) is the weAutSys implementation of protothreads.

2.11.2.7 #define INFLASH(*decl*)

Alternative declaration for flash memory.

This macro is a replacement for the use of the PROGMEM macro.

One can end an `INFLASH` declaration by just a (;) semicolon. But as it is for final flash / program memory variables a declaration is usually accompanied by an initialiser, like in:

```
INFLASH(char flashString[]) = "in program memory and only there";
```

Most C implementations, like the GNU C Compilers and tools (GCC) used here, can't handle Harvard architecture in a smooth consistent and portable way. Instead of healing that, ugly helper constructs (<avr/pgmspace.h>) have been invented and, alas, must be used. Among other problems their usage then causes tooling problems (Doxygen, Eclipse).

Internal: In case of macro **DOXYGEN** defined INFLASH just leaves `decl` without extra attribute. (Doxygen hates PROGMEM.)

Hint: The pgm_read_byte(address) expression has to be used to access INFLASEHed final variables. This works only in the range 0..64k-1. If INFLASEHed items happen to life above 64k pgm_read_byte_far(longByteAddress) has to be used. longByteAddress will not be constructed correctly by avr-gcc C pointer handling in most cases.

Parameters

decl	a (standard), i.e. "type name", variable declaration without further modifiers

See also

INEEPROM

Examples:

main.c.

2.11.2.8 #define INEEPROM(*decl*)

Declaration for EEPROM memory.

This macro is used to define variables in the EEPROM usually for generating individualised .hex files for configuration settings (like MAC address etc.) If, for example, the source individEEP.c contains such settings to be in the EEPROM starting at address 0x80 one would do something like this:

```
 avr-gcc -c -mmcu=atmega1284p -Wl,--section-start=.eeprom=0x0080 -I.  -I
./include individEEP.c
 avr-objcopy -j .eeprom --change-section-lma .eeprom=128 --no-change-
warnings -O ihex individEEP.o  individEEP.hex
 avrdude -p atmega1284p -c avrisp2  -P com4   -U eeprom:w:individEEP.hex:
i
```

One can end an `INEEPROM` declaration by just a (;) semicolon. But as it is in the end for generating final values for (individualised) EEPROM .hex files the variables defined are usually accompanied by an initialiser, like in the example:

```
INEEPROM(uint8_t redundantEndMarker) = 0x77;
```

Parameters

decl	a (standard), i.e. "type name", variable declaration without further modifiers

See also

INFLASH

Examples:

individEEP.c.

2.11.2.9 #define **STR(** *leMac* **)**

A macro value as string.

This macro implements the "stringification" of a marco's value. See gnu.org's GCC documentation (chapter 1.4.4). GCC's stringification has some special rules concerning white space and escapes. Hence for more complicated values the result might not meet the expectations.

Parameters

leMac	the macro to stringify

2.11.2.10 #define **FCPU_S**

The CPU clock frequency (as string)

See the macro `F_CPU` defined in the `makefile`.

This is its value as string, normally "20000000" for 20MHz.

2.11.2.11 #define **PLATFORM_S**

The platform name (as string)

See the macro (respectively make variable) `PLATFORM` defined in the `makefile`.

This is its value as string, that is "weAut_01" and the like.

2.11.2.12 #define **SYST_AUT** "Albrecht Weinert <a-weinert.de> "

The runtime's author.

It is the author's name and his personal domain.

The value is a "quoted" string.

2.11.2.13 #define SYST_COP

Value:

```
"Copyright (c) 2014  weinert - automation  \n"\
                "Prof. Dr.-Ing. Albrecht Weinert,  Bochum  \n"
```

The runtime's copyright notice.

The value is a two line "quoted" string with trailing linefeed.

2.11.2.14 #define SYST_BLD

The runtime's system build and time.

It is the time this file was processed by the compiler — or more correct by the pre-processor.

The value is a "quoted" string.

2.11.2.15 #define HBYTE(*x*)

Get the high byte (of a 16 bit value)

This is a helper. It gets second byte of a value (larger than one byte). That would be the high byte of a 16 bit value.

It is expressed by masking the high byte and shifting right by 8. The AVR C Compiler is clever enough to just pick the appropriate byte respectively register as result. So this helper does not harm.

2.11.2.16 #define LBYTE(*x*)

Get the low byte (of a 16 bit value)

This is a helper that just masks the lowest byte of a 16 bit (or larger) value.

See also: HBYTE

2.11.2.17 #define OFF

A boolean false respectively off.

The value of false / off / aus / éteint is 0.

Examples:

main.c.

2.11.2.18 #define ON

A boolean true respectively on.

The value of true / on / an / démarré is `FF`... meaning all bits one for any size or type.

Examples:

main.c.

2.11.2.19 #define LF

The linefeed code.

A LF character is usually the equivalent of \n. LF and CR are counted in the serial input buffer.

The value of `LF` is 10 respectively 0x0A

2.11.2.20 #define CR

The carriage return code.

See also

LF The value of `CR` is 13 respectively 0x0D

2.11.2.21 #define BEL

The bell code.

A BEL character is produces an audible (or visible) signal but no visible / printable character output (not even white space).

BEL my be used by some Telnet software.

The value of `BEL` is 7

2.11.2.22 #define BS

The back space code.

BS may be used by some Telnet software to [rfc854]:

moves the (NVT printer's) print head one character position towards the left margin.

The value of `BS` is 8

2.11.2.23 #define HT

The horizontal tab code.

HT may be used by some Telnet software to [rfc854]

move the (NVT printer's) print head to the next horizontal tab stop. It remains unspecified how either party determines or establishes where such tab stops are located. [rfc854 cite end]

The value of `HT` is 9

2.11.2.24 #define VT

The vertical tab code.

VT may be used by some Telnet software to [rfc854]

move the (NVT printer's) print head to the next vertical tab stop. It remains unspecified how either party determines or establishes where such tab stops are located. [rfc854 cite end]

The value of `VT` is 11 (0x0B)

2.11.2.25 #define FF

The form feed code.

FF may be used by some Telnet software to [rfc854]

move the (NVT printer's) print head to the top of the next page, keeping the same horizontal position. [rfc854 cite end]

The value of `FF` is 12 (0x0C)

2.11.2.26 #define ESC

The Escape code.

ESC may be used as a sync character e.g. by some bootloader software.

The value of `ESC` is 0x1B

2.11.2.27 #define APP_END

The end of the application flash area (as byte address + 1)

This value is the end of the Read-While-Write (RWW) section in flash + 1 as byte address. For an ATmega 1284 this is e.g.:

122880 = 1E000 = F000 $*$ 2

For an ATmega 2560 this is e.g.:

253952 = 3E000 = 1F000 $*$ 2

The usage of byte addresses forces more than 16 bit for the address in this case.

The value is ((FLASHEND - BOOTSIZE) + 1)

2.11.2.28 #define BOOTL_BEG

The start address of the bootloader (area)

This byte address in flash memory is just APP_END.

2.11.2.29 #define LARGE_MEMORY 0

The flash memory byte address type.

The ATmega's flash memory is a word (16bit) organised program memory in Harvard architecture. Hence a 16 bit program counter is sufficient for a 128 kByte flash (ATmega1284 e.g.). Nevertheless final variables can be placed in flash, too, forcing it to be byte addressable therefore. In that sense `LARGE_MEMORY` true (not 0) means we need byte addresses larger than 16 bit forcing the address type from 16 to 32 bit.

Note: 24 bit would suffice for all large ATmegas, but C hardly handles that size.

See also

ADDR_T

2.11.3 Typedef Documentation

2.11.3.1 typedef ptfnct_t(funV_t)(void)

Type of a protothread function (void)

`funV_t` is a function returning PT_WAITING, PT_YIELDED, PT_EXITED or PT_ENDED and taking no parameter.

2.11.3.2 typedef funV_t∗ p2ptFunV

Pointer to a protothread function (void)

`p2ptFunV` is a pointer to a function returning PT_WAITING, PT_YIELDED, PT_EXITED or PT_ENDED and no parameter.

2.11.3.3 typedef ptfnct_t(funU_t)(struct mThr_data_t ∗uthr_data)

Type of a protothread function (struct mThr_data_t ∗)

`funU_t` is a function returning PT_WAITING, PT_YIELDED, PT_EXITED or PT_ENDED and taking a pointer to the basic minimal thread state (mThr_data_t) as parameter.

2.11.3.4 typedef funU_t∗ p2ptFun

Pointer to a protothread function (struct mThr_data_t ∗)

`p2ptFun` is a pointer to a function returningPT_WAITING, PT_YIELDED, PT_EXITED or PT_ENDED and taking a pointer to the basic minimal thread state (mThr_data_t) as parameter.

2.11.3.5 typedef ptfnct_t(fun_t)(struct thr_data_t ∗thrData)

Type of a protothread function (struct thr_data_t ∗)

`fun_t` is a function returning PT_WAITING, PT_YIELDED, PT_EXITED or PT_ENDED and taking a pointer to the common variable type thread state as parameter.

2.11.3.6 typedef fun_t∗ p2ptFunC

Pointer to a protothread function (struct thr_data_t ∗)

`p2ptFunC` is a pointer to a function returning PT_WAITING, PT_YIELDED, PT_EXITED or PT_ENDED and taking a pointer to the common variable type thread state (thr_data_t) as parameter.

2.11.3.7 typedef ptfnct_t(funA_t)(struct cliThr_data_t ∗thrData)

Type of a protothread function (cliThr_data_t ∗)

`funA_t` is a function returning PT_WAITING, PT_YIELDED, PT_EXITED or PT_ENDED and taking a pointer to a CLI (thread) state (cliThr_data_t) as parameter.

2.11.3.8 typedef funA_t∗ p2ptFunA

Pointer to a protothread function (cliThr_data_t ∗)

`p2ptFunA` is a pointer to a function returning PT_WAITING, PT_YIELDED, PT_EXITED or PT_ENDED and taking a pointer to a CLI (thread) state (cliThr_data_t) as parameter.

2.11.4 Variable Documentation

2.11.4.1 char const l4nefeeds[ ]

A short string with just one blank and four linefeeds in flash memory.

This flash string as all cognates (bLF1, bLF2 and many more) are 0 terminated. As flash strings they are immutable (by all normal and erroneous program means). They should be used whenever their content is asked for to save precious RAM.

2.12 Command line interpreter (CLI)

2.12.1 Overview

weAutSys has an command line interpreter (CLI or shell) that parses and executes some 30 system commands with a rich set of options.

User / application software may easily add (up to 100) user specific commands plus an own interpreter. It has to be registered with the weAutSys runtime. Then all user commands will be delegated to it — pre-parsed and prepared.

This CLI or shell may — by user software initialisation — be used with

- serial communication (UART),
- the Telnet or
- other links

in any combination and (quasi) parallel in multiple threads.

Files

- file cli.h
  weAutSys' command line interpreter (CLI)

Data Structures

- struct appCLIreg_t
  The user / application CLI registration type.

Defines

- #define YIELD_FOR_BUSY_CLI(appThread)
  Wait (yielding) for command execution to end.

Functions

- ptfnct_t appCliThreadF (struct cliThr_data_t ∗cliThread)
  The user / application specific command line interpreter thread.
- void initAsCLIthread (struct thr_data_t ∗thread, FILE ∗file)
  Initialise a thread (structure) as command line interpreter (CLI) thread.
- void registerAppCli (p2ptFunA threadF, char const ∗const userCommands[ ])
  Register a user / application command line interpreter (CLI)
- uint8_t setCliLine (struct cliThr_data_t ∗const cliThread, char ∗const line, uint16_t length)
  Set a command line in the command line interpreter (CLI) thread data.
- ptfnct_t sysCliThreadF (struct cliThr_data_t ∗cliThread)
  The system command line interpreter thread.
- void unimplOptionReport (struct cliThr_data_t ∗cliThread)
  Report the use of an un-implemented option for a command.

Variables

- struct appCLIreg_t appCLIreg
  The user / application command line interpreter (CLI) registration.

2.12.2 Define Documentation

2.12.2.1 #define YIELD_FOR_BUSY_CLI(*appThread*)

Wait (yielding) for command execution to end.

This macro waits yielding the main (input, application) thread for all command interpretation to end. It is a helper for a user defined serial input thread.

Parameters

appThread	pointer to the (complete, type thr_data_t) thread structure

Examples:

main.c.

2.12.3 Function Documentation

2.12.3.1 void registerAppCli (p2ptFunA *threadF*, char const ∗const *userCommands[]*)

Register a user / application command line interpreter (CLI)

The user CLI thread function and the user commands description will be registered.

To de-register call with at least one parameter NULL.

Parameters

threadF	the user CLI thread function to register
userCommands	flash array of flash strings of user commands (+ explanations). The (non NULL) array must end with NULL entry.

See also

appCliThreadF
appInitThreadF

Examples:

main.c.

2.12.3.2 void initAsCLIthread (struct thr_data_t ∗ *thread*, FILE ∗ *file*)

Initialise a thread (structure) as command line interpreter (CLI) thread.

This function initialises the common (union) part of the structure supplied as a command line interpreter (CLI) thread for the usage of a stream (`file`) for output and the registered application command line interpreter (CLI) thread function.

The (CLI) thread state will completely be forced to empty initial state with no command and line set.

Parameters

thread	the thread structure to initialise
file	The streams to be used for output / reply; NULL is OK when all command may be executed (or even fail) silently

See also

appCliThreadF
appInitThreadF

Examples:

main.c.

2.12.3.3 uint8_t setCliLine (struct cliThr_data_t ∗const *cliThread*, char ∗const *line*, uint16_t *length*)

Set a command line in the command line interpreter (CLI) thread data.

The given `line` will be set in the `cliThread`, the command (i.e. the first) token will be determined and the system and user CLI thread state will be forced to empty initial state.

Do not call this function while system or user CLI threads are active.

The `line` (buffer) is copied to have it as thread state. The copying and extra memory usage is avoided if the parameter `line` is prepared in and supplied as `cliThread->cliThrData.line`.

Parameters

cliThread	pointer to the CLI thread to interpret line
line	the command line (pointer to 0-terminated RAM string)
length	`line's` length

Returns

the command number

See also

appCliThreadF
appInitThreadF

Examples:

main.c.

2.12.3.4 ptfnct_t appCliThreadF (struct cliThr_data_t ∗ *cliThread*)

The user / application specific command line interpreter thread.

If something is to be done in the user / application software for (human readable) command lines (set by setCliLine()) a command line interpreter protothread thread function must be provided therefore.

This has to be done (according to this declaration) in an application specific source file and that thread function has to be registered.

This thread function will be scheduled automatically if and as long as a user command is pending. If an output stream belongs to the calling context the standard streams will have been switched to it space for 40 characters is guaranteed. So this user thread function's implementation will not have to worry about those house keeping tasks.

This thread function will be re-scheduled on a pending user command as long as the command (`cli-Thread->commNumb`) is not set to 0 or this function does not ends or exits.

See also

initAsCLIthread
appInitThreadF
sysCliThreadF

2.12.3.5 ptfnct_t sysCliThreadF (struct cliThr_data_t * *cliThread*)

The system command line interpreter thread.

If a (human readable) command line has been set by setCliLine() this thread might be scheduled (until end / exit) to execute a parsed system command.

If user commands and a user CLI thread (function) are registered this thread will also be scheduled if a user command was parsed. This thread function will delegate all user command execution to the registered user user CLI thread (function)

If no (unambiguous) command was found a report will be made if allowed so in the `cliThread` structure.

See also

initAsCLIthread
applnitThreadF
appCliThreadF

Returns

PT_YIELDED if waiting for conditions that may become automatically true; PT_WAITING if blocked by conditions that other's action may set true; PT_EXITED if stopped by conditions that never will become true (sorry, not all work done and never will; PT_ENDED if ready with all work

2.12.3.6 void unimplOptionReport (struct cliThr_data_t * *cliThread*)

Report the use of an un-implemented option for a command.

This is a helper function for the implementation on CLIs (command line processors) only. This function requires stdout connected to CLI streams and enough output space available in the respective stream.

Parameters

cliThread	pointer to the CLI (and hence the thread busy to execute actual "command -wrongOpt)

2.12.4 Variable Documentation

2.12.4.1 struct appCLIreg_t appCLIreg

The user / application command line interpreter (CLI) registration.

Command line interpretation can occur in multiple contexts like UART link, multiple connections to Telnet server etc. And it can occur "simultaneously" in in any of said contexts.

Nevertheless the application CLI (if any) is registered here for once. This structure with this registration may and will be copied to the state / structure of said usages / connections.

It may take some time and (disconnect / connect) events for a changed application CLI registration to take effect. Or, in other words dynamic changes of an application CLI aren't supported yet. It is strongly recommended for the user software to have zero or one application CLI implementation and register it at reset start-up (phase 4).

2.13 NTP client

2.13.1 Overview

At the upper stack levels uIP brings some protocol and application support. weAutSys adapted and uses some of those (DHCP, ARP, DNS) as well as implementing some other protocols (like NTP, Telnet server).

weAutSys' NTP client is best used with above said DHCP client, which can get one or two NTP server addresses. Any Windows or appropriate Linux server on the network segment can take the role of the time server for all (weAut_01) automation modules there.

DHCP and NTP may very well be on the same server machine.

As the recommended configuration is to have a separated private LAN segment for all automation modules plus their supporting central servers some of the complexities of the NTP protocol (with the RFC's permission) are dismissed.

Complexities dismissed are, for example, expensive filtering and selection algorithms which could deal with malicious servers among others. It is assumed Bycantine servers being kept away from above said automation LAN by other means. And of course, all principal limitations of NTP synchronisation, like unavoidable offsets due to network round trip and software time stamping asymmetries etc., do apply also here.

On the other hand weAutSys' NTP client implementation is not as simplistic as SNTP. It uses the obvious NTP difference formulas and sets the date / time (hard) for big differences. Relative times and timers are not affected by those settings ("jumps"), but functions and timers dependent on absolute time and date may be.

Such hard setting of date and time will occur once after reset or power up as soon as a connection to a NTP server is established.

Small differences will be caught up by slightly trimming the oscillator responsible for the milliseconds tick. As long as the network and above said NTP server behave well, the control strategy implemented will make this tick exact in the median. And it will avoid all further (hard) date / time settings. The only exception then remaining will necessarily be leap seconds and DST shifts.

Relative milliseconds and seconds timers will (in the median) also get the NTP servers accuracy. Depending on atomic time, NTP time and related frequency accuracy is better than weAut_01 quartz oscillator's — which is quite good by the way. But beware, there are reports of surprisingly bad (black sheep) NTP servers around — using (indirectly) on of them is worse than having none. Just check.

Files

- file ntp.h

  weAutSys' system calls and types for network time services

Data Structures

- struct ntpMess_t

  NTP message type.
- struct ntpState_t

  Structure for NTP (client) application state.
- struct ntpTimestamp_t

  NTP time stamp type.

Defines

- #define FLAGS_LI_MASK

  leap indicator bits in ntpMess_t::flags
- #define FLAGS_MODE_MASK

  mode bits in ntpMess_t::flags

- #define FLAGS_VERSION_MASK

    version bits in ntpMess_t::flags
- #define LI_59SEC

    leap indicator 2 : minute = 59s
- #define LI_61SEC

    leap indicator 1 : minute = 61s
- #define LI_ALARM

    leap indicator 3 : alarm (clock not synchronized)
- #define LI_NOWARN

    leap indicator 0 : no warning
- #define NTP_ADJUSTAT_DIRMSK

    last adjustment direction mask
- #define NTP_ADJUSTAT_MINUS

    last adjustment negative
- #define NTP_ADJUSTAT_PLUS

    last adjustment positive
- #define NTP_ADJUSTAT_PRV_MINUS

    previous (not last) adjustment negative
- #define NTP_ADJUSTAT_PRV_MSK

    previous (not last) adjustment mask
- #define NTP_ADJUSTAT_PRV_PLUS

    lprevious (not last) adjustment positive
- #define NTP_ADJUSTAT_RNG_1S

    last adjustment range +1 second
- #define NTP_ADJUSTAT_RNG_HM

    last adjustment range milliseconds
- #define NTP_ADJUSTAT_RNG_HS

    last adjustment range seconds
- #define NTP_ADJUSTAT_RNG_LM

    last adjustment range low milliseconds
- #define NTP_ADJUSTAT_RNGMSK

    last adjustment range mask
- #define NTP_BROADCAST

    NTP mode broadcast (5)
- #define NTP_CLIENT

    NTP mode client (3)
- #define NTP_FRACT_DROP

    Fraction byte significance.
- #define NTP_PORT

    The well known NTP port.
- #define NTP_SERVER

    NTP mode server (4)
- #define NTP_STATE_CNETAB 0xE0

    connection established
- #define NTP_STATE_CNPROB 0xF0

    connection problematic
- #define NTP_STATE_CONN1 1

    Server known, port connected to server 1.
- #define NTP_STATE_CONN2 2

    Server known, port connected to server 2.
- #define NTP_STATE_CONTSY 0xC0

continuous synchronisation
- #define NTP_STATE_OPMSK

  Operation (bits) mask.
- #define NTP_STATE_RESET 0

  No known NTP server, no NTP client operation.
- #define NTP_STATE_SEPROB 0xB0

  server answer problematic (bogus, alarm)
- #define NTP_STATE_SRVMSK

  Server (bits) mask.
- #define NTP_STATE_TRYREQ 0x10

  try (first) request
- #define NTP_STATE_W4REPL 8

  flag: waiting for server's reply
- #define NTP_VERSION

  current version (4) within FLAGS_VERSION_MASK
- #define ntpAsksPrio()

  NTP asks for priority.

Functions

- void adjustNTPtime308UTC (struct ntpTimestamp_t ∗ntpTimeDif)

  Adjust the actual (system) time by a NTP time stamp difference.
- ptfnct_t ntpAppcall (void)

  Handle NTP (server) events.
- void ntpInit (uint8_t pref2)

  Initialise the NTP (client)
- void ntpReset (void)

  Reset the NTP (client) to initial state.
- void ntpTimestampDif (struct ntpTimestamp_t ∗result, struct ntpTimestamp_t ∗a, struct ntpTimestamp_t ∗b)

  Difference of two NTP time stamps.
- void ntpTimestampHalf (struct ntpTimestamp_t ∗result) __attribute__((pure))

  Half a NTP time stamp value.
- void ntpTimestampSum (struct ntpTimestamp_t ∗result, struct ntpTimestamp_t ∗a, struct ntpTimestamp_t ∗b)

  Sum of two NTP time stamps.
- void ntpTimestampToMillies (uint16_t ∗ms, struct ntpTimestamp_t ∗ntpStamp)

  Milliseconds from a NTP time stamp's fraction.
- void setNTPstampAct (struct ntpTimestamp_t ∗ntpTimestamp)

  Set a NTP time stamp to actual time.

Variables

- struct ntpState_t ntpState

  the NTP state

2.13.2 Define Documentation

2.13.2.1 #define NTP_PORT

The well known NTP port.

NTP normally uses UDP port 123.

2.13.2.2 #define NTP_FRACT_DROP

Fraction byte significance.

As said NTP time stamp's fraction has four bytes allowing 2∗∗-32 = ∼0.23ns resolution. That is far beyond any practical use for e.g. ATmel ATmega1284P based automation modules, like weAut_01 with 20MHz processor clock (50ns period) at best.

Hence this value determines the number of bytes being ignored for all 64 bit NTP time stamp operations.

Range: 0 (full 64 bit arithmetic) .. 3 (fraction is 8 bit only)

Recommended / default: 2 (16 bit fraction, sub-ms resolution)

The recommended value 2 is a good choice also in the light of most NTP servers. Observations of Windows 2008 R2 as NTP server showed the least 16 fraction bits being always the same (rubbish ?) value no matter the circumstances.

`NTP_FRACT_DROP` 2 reduces ntpTimestampSum, ntpTimestampDif and ntpTimestampHalf() from 64 bit to 48 bit arithmetic (saving ten / six cycles). `NTP_FRACT_DROP` 2 also fits well to ntpTimestampToMillies() and setNTPstampAct().

2.13.2.3 #define ntpAsksPrio( )

NTP asks for priority.

This expression is true if the NTP client sent a request and waits for the response. Postponing the reaction to this response by using needed resources (NIC, SPI) or having long running thread steps (by SMC block operations e.g.) will detriment the synchronisation accuracy.

As NTP synchronisations are quite seldom (about every minute in good sync state) and turn-around for (recommended) local NTP servers is quite fast it should be feasible for all others to respect this request for priority.

2.13.3 Function Documentation

2.13.3.1 void ntpTimestampSum (struct ntpTimestamp_t ∗ *result,* struct ntpTimestamp_t ∗ *a,* struct ntpTimestamp_t ∗ *b*)

Sum of two NTP time stamps.

This function calculates ∗result = ∗a + ∗b;

No pointer must be null, but two or three of them might point to the same ntpTimestamp_t structure.

ntpTimestampSum, ntpTimestampDif and ntpTimestampHalf are optimised implementations for 64 bit wrong endian (i.e. network byte order).

The former two take 74 cycles and the latter 43. This values are for NTP_FRACT_DROP 0, the recommended value 2 makes all a bit faster.

Parameters

a	points to first operand (not NULL)
b	points to second operand (not NULL)
result	points to result structure (not NULL)

See also

NTP_FRACT_DROP

2.13.3.2 void **ntpTimestampDif** (struct **ntpTimestamp_t** ∗ *result,* struct **ntpTimestamp_t** ∗ *a,* struct **ntpTimestamp_t** ∗ *b*)

Difference of two NTP time stamps.

This function calculates ∗result = ∗a - ∗b;

No pointer must be null, but two of them might point to the same ntpTimestamp_t structure.

ntpTimestampSum, ntpTimestampDif and ntpTimestampHalf are optimised implementations for 64 bit big endian.

Parameters

a	points to first operand (not NULL)
b	points to second operand (not NULL)
result	points to result structure (not NULL)

See also

NTP_FRACT_DROP

2.13.3.3 void **ntpTimestampHalf** (struct **ntpTimestamp_t** ∗ *result*)

Half a NTP time stamp value.

This function calculates ∗result = ∗result / 2

The division by 2 is performed for a signed (two complement) 64 bit number. This also gives correct results for large differences in modulo 2∗∗64 bit arithmetic, but will of course fail on very large unsigned numbers >= 2∗∗63.

ntpTimestampSum, ntpTimestampDif and ntpTimestampHalf are optimised implementations for 64 bit big endian.

Parameters

result	points to operand and result structure (not NULL)

See also

NTP_FRACT_DROP

2.13.3.4 void **ntpTimestampToMillies** (uint16_t ∗ *ms,* struct **ntpTimestamp_t** ∗ *ntpStamp*)

Milliseconds from a NTP time stamp's fraction.

This function converts a NTP time stamp's fraction value (32 bit) into milliseconds in the range 0 .. 999 (16 bit).

The fraction is expected in (wrong) network endianess while the result will be set in the correct byte order.

The implementation is (inline) ASM optimized. It sacrifices a tiny bit of exactness in favor of a slow RISC processor with only byte operations. The effect is the 0 .. 999ms being mapped to 0..992ms.

The assumption behind is the worst case 1% of a second error a being far within the NTP synchronisation abilities of an ATmega-uIP-ENC trio.

Round trip times of 50ms .. 80ms with a Windows 2008 R2 time server in a local network with some switches in between suggest the un-symmetry under these near ideal conditions being at least some milliseconds. Hence the above assumption seems quite adequate.

Parameters

ntpStamp	structure pointed to has fraction in network byte order; never null! Using a pointer for this 16 bit result instead of a return value is just a work around a GCC inline ASM bug.
ms	points to the result variable where to put value.fraction converted to ms (0 .. 999) to; never null

See also

ntpTimestampSum
ntpTimestampHalf
setNTPstampAct

2.13.3.5 void setNTPstampAct (struct ntpTimestamp_t ∗ *ntpTimestamp*)

Set a NTP time stamp to actual time.

This function sets the .seconds and .fraction of `ntpTimestamp` to the actual UTC system time (secTime308UT-C(), msAbsClockCount).

On .fraction all but the upper 10 bits are set to zero.

The ms to fraction algorithm implementation is (ASM) optimised for the 8bit AVR RISC. It sacrifices a tiny bit of exactness in favor of great CPU usage savings. The effect is the 0 .. 999ms of msAbsClockCount being mapped to effectively 0..992ms in the NTP stamp fraction.

Parameters

ntpTimestamp	points to the stamp to be set (never null)

See also

ntpTimestampToMillies

2.13.3.6 void adjustNTPtime308UTC (struct ntpTimestamp_t ∗ *ntpTimeDif*)

Adjust the actual (system) time by a NTP time stamp difference.

This function adjusts the internal date and time (secTime308UTC()) by an amount given as a a NTP time stamp difference to theactual time in s, ms.

Big (>=2∗∗31 in the seconds part) differences will quite naturally act as negative in the sense of second based timing's (and NTP's) modulo 2∗∗32 arithmetic.

This function should be used for any absolute date time adjustments caused by NTP client functions (or may be also other time sources).

The action taken and the flags set in ntpState .lastAdjust depend on the (absolute) difference amount

- below 8 ms : no action (considered as jitter)
- -100 .. +50ms : no setting;

  only oscillator speed up / slow down
- -999 .. + 999ms : setting of time and

  oscillator speed up / slow down
- above -1 .. +1s : setting of time only

Parameters

ntpTimeDif	the adjustment difference in seconds,fraction; big values act as negative in the sense of seconds modulo 2∗∗32 arithmetic

See also

secTime308Loc(void)
secTime308UTC(void)

2.13.3.7 void ntpInit (uint8_t *pref2*)

Initialise the NTP (client)

This function initialises the NTP structures and (protothread) state machine. The protocol is not started by this function; the work is done by later calls of ntpAppcall(). There is no need to call ntpReset() before.

If "be NTP client" is not set in curIpConf nothing will be done except setting the state to NTP_STATE_RESET.

If the parameter `pref2` is true (not 0) the second NTP server in the current IP configuration will be used, if set there. Successful bind to a set NTP server will set the (NTP client) state to NTP_STATE_CONN1 respectively NTP_STATE_CONN2.

Parameters

pref2	if true use second NTP server (if available)

2.13.3.8 void ntpReset (void)

Reset the NTP (client) to initial state.

This function initialises the NTP structures and (protothread) state machine. This function resets all NTPP state to initial. In contrast to ntpInit no trial to connect or bind to NTP ports or servers. The protocol is not (not even indirectly) started.

See also

ntpInit()

2.13.3.9 ptfnct_t ntpAppcall (void)

Handle NTP (server) events.

This function handles NTP events. And it is a (as a protothread) the NTP client state machine. It has to be called in the udp_appcall in a manner like

```
const uint16_t remPort = convert16endian(uip_udp_conn->rport);

if(remPort == NTP_PORT) {   // NTP protocol
   ntpAppcall();
   return;
} // NTP protocol
```

2.14 Telnet server

2.14.1 Overview

At the upper stack levels uIP brings some protocol and application support. weAutSys adapted and uses some of those (DHCP, ARP, DNS) as well as implementing some other protocols (like Telnet, NTP client).

weAutSys' Telnet server implementation exposes the system and (optionally) user software command line interpreter to the Ethernet. This can free the UART (also usable for CLI) for other purposes. Up to the common maximum connection number multiple CLI clients — plus one at the UART — could be served.

In favour of the embedded controller's resources and communication load this Telnet server assumes respectively insists on (kludge) line mode. So, not much is done on mode negotiation and nothing for special client side terminals.

For the same reasons this Telnet server does nothing on encryption or authentication. This is justified by the assumed configuration having a separated private LAN segment for all automation modules plus their supporting central servers and HMI stations. Should one of these servers bridge the automation LAN to the outside world it must take the security burden.

To find Telnet clients that could handle the said simple line mode without causing any troubles or complications proved surprisingly difficult for a variety of platforms. Using the `Frame4J` tool `ClientLil` by

```
java ClientLil -telnet 192.168.89.343 -v
```

on any (Windows, Linux etc.) platform with a standard JDK/JRE (1.6.0 or higher) with `Frame4J` (`jar` 1.07.02 or higher) as `installed` extension.

Files

- file telnet.h

  weAutSys' system calls, services and types for the Telnet server

Defines for Telnet command, state and options

This is a subset of the command and option codes defined for the Telnet protocol.

The use of the others not put here is discouraged.

- #define IAC

  Telnet: interpret as command (IAC)
- #define DONT 254

  Telnet: option negotiation command respectively answer.
- #define DO 253

  Telnet: option negotiation command respectively answer.
- #define WONT 252

  Telnet: option negotiation command respectively answer.
- #define WILL 251

  Telnet: option negotiation command respectively answer.
- #define LINEMODE 34

  Telnet option code.
- #define SUPPRESS_GA 3

  Telnet option suppress go ahead.
- #define TEL_SB 250

  Telnet command opening brace SB.
- #define TEL_SE 240

  Telnet command closing brace SE.

Functions

- ptfnct_t telnetAppcall (void)

    Handle Telnet server events.
- void telnetInit (void)

    Initialise the Telnet (server)

2.14.2 Define Documentation

2.14.2.1 #define IAC

Telnet: interpret as command (IAC)

All Telnet commands consist of a sequence of at least a two bytes:

1. the "Interpret as Command" (IAC) escape character
2. the code for the command.

The commands dealing with option negotiation are three byte sequences:

1. the "Interpret as Command" (IAC) escape character
2. the code for the command
3. the code for the option referenced in command.

Sub-negotiation are at least five bytes long and are structured as:

IAC SB information byte(s) IAC SE

The value of IAC is 255 or 0xFF.

The value will also be used as (Telnet command interpreter) state.

If `0xFF` is part of the data it has to be escaped (by doubling it). I.e. IAC IAC does not act as command but as normal data byte `0xFF`.

2.14.2.2 #define DONT 254

Telnet: option negotiation command respectively answer.

`DONT` demands (other) no longer to perform indicated option. The positive acknowledge is a corresponding WONT command.

The value will also be used as (Telnet command interpreter) state.

2.14.2.3 #define DO 253

Telnet: option negotiation command respectively answer.

`DO` requests indicated option. The positive acknowledge is WILL the negative answer is WONT.

The value will also be used as (Telnet command interpreter) state.

See also: DONT

2.14.2.4 #define WONT 252

Telnet: option negotiation command respectively answer.

`WONT` refuses (to continue) the indicated optiond. The positive acknowledge is a corresponding DONT command.

The value will also be used as (Telnet command interpreter) state.

2.14.2.5 #define WILL 251

Telnet: option negotiation command respectively answer.

`WILL` wants or confirms the indicated option. The positive acknowledge is DO the negative answer is DONT.

The value will also be used as (Telnet command interpreter) state.

See also: WONT

2.14.2.6 #define TEL_SB 250

Telnet command opening brace SB.

IAC `SB` indicates the begin of sub-negotiation data.

The value will also be used as (Telnet command interpreter) state.

See also: SE

2.14.2.7 #define TEL_SE 240

Telnet command closing brace SE.

IAC `SE` indicates the end of sub-negotiation data and the sub-negotiation command as whole.

See also: SB

2.14.3 Function Documentation

2.14.3.1 void telnetInit (void)

Initialise the Telnet (server)

This function initialises the Telnet server. It is (indirectly) called whenever the Ethernet link goes up and hence usually after reset.

At present the implementation does nothing, but that may change in future releases.

2.14.3.2 ptfnct_t telnetAppcall (void)

Handle Telnet server events.

This function handles all Telnet events. And it is a as a protothread (part of) the Telnet server state machine.

This Telnet server implements the basic communication, a CLI (command line interpreter) and a log buffer consumer. The appstate part of the connection state hence is thread data of type thr_data_t in the cliThr_data_t variant.

The current appstate.flag usage is:

Bit 0 (1): command interpreter active

Bit 1 (2): prompt output (after last output) pending

Bit 3 (4): consume and send the pending bufLogStreams output (before prompt).

2.15 Modbus server

2.15.1 Overview

At the upper stack levels uIP brings some protocol and application support. weAutSys adapted and uses some of those (DHCP, ARP, DNS) as well as implementing some other protocols.

weAutSys' Modbus server implementation is a comfortable hook for user / application software to expose process and / or other values for reading and writing via Modbus TCP using that well known open protocol published 1979.

Files

- file modbus.h

  weAutSys' system calls, services and types for the Modbus server

Data Structures

- struct modConfData_t

  The configuration data (type) for a Modbus handling.
- struct modTelegr_t

  The (start of a) Modbus TCP/IP telegram.
- struct modThr_data_t

  The organisational data for a Modbus handler thread.

Defines

- #define COIL_SINGLE

  Modbus data model: Single coil bit addressed 'C'.
- #define COILS_BYMAPD

  Modbus data model: Coils byte mapped 'c'.
- #define DISC_INP_BYMAPD

  Modbus data model: Discrete inputs byte mapped 'i'.
- #define HLD_MASK

  Modbus data model: Single Holding register for mask 'H'.
- #define HLD_REGISTERS

  Modbus data model: Holding registers 'h'.
- #define INP_REGISTERS

  Modbus data model: Input registers 'r'.
- #define MODB_FCIND 7

  Modbus: index of function code in the TCP/IP telegram.
- #define MODB_MBAB_LEN 7

  Modbus: length of the TCP/IP telegram start (MBAB header)

Defines for Modbus function codes and the like

This is a subset of the function codes defined for the Modbus protocol.

The others not put here are not implemented by this server.

- #define WRITE_COIL 0x05

  Modbus function code: write one bit output.

- #define READ_COILS 0x01

  Modbus function code: read back bitwise output.
- #define WRITE_COILS 0x0F

  Modbus function code: write bitwise output.
- #define READ_DISCRETE_INPUTS 0x02

  Modbus function code: read bitwise inputs.
- #define READ_INPUT_REGISTERS 0x04

  Modbus function code: read word (16 bit) inputs.
- #define READ_HOLDING_REGISTERS 0x03

  Modbus function code: read back word (16 bit) outputs.
- #define WRITE_HOLDING_REGISTERS 0x10

  Modbus function code: write word (16 bit) outputs.
- #define WRITE_HOLDING_REGISTER 0x06

  Modbus function code: write (one) word (16 bit) output.
- #define MASK_WRITE_REGISTER 0x16

  Modbus function code: and and or (one) word (16 bit) output.
- #define WRITE_READ_REGISTERS 0x17

  Modbus function code: write than read word (16 bit) outputs.
- #define MODB_EXC_FUNC 1

  Modbus exception: unimplemented function code.
- #define MODB_EXC_ADDR 2

  Modbus exception: illegal address.
- #define MODB_EXC_DATA 3

  Modbus exception: invalid data (or length)
- #define MODB_EXC_OPER 4

  Modbus exception: the (partly) performed operation failed.

Typedefs

- typedef ptfnct_t(funM_t)(struct modThr_data_t ∗thrData)

  Type of a protothread function (modThr_data_t ∗)
- typedef funM_t ∗ p2ptFunM

  Pointer to a protothread function (modThr_data_t ∗)

Functions

- ptfnct_t appModFun (struct modThr_data_t ∗m)

  Handle Modbus server events.
- ptfnct_t modbusAppcall (void)

  Handle Modbus server events.
- void modbusInit (void)

  Initialise the Modbus (server)
- void registerAppModFun (p2ptFunM appModFun)

  Register the application Modbus handler function.

2.15.2 Define Documentation

2.15.2.1 #define WRITE_COIL 0x05

Modbus function code: write one bit output.

This function is to write one single bit output. The lower 3 bits in the target address would be the bit number within a byte and the address / 8 would be the byte number if mapped to bytes.

2.15.2.2 #define READ_COILS 0x01

Modbus function code: read back bitwise output.

The Modbus standard considers this function code's address as bit number and the quantity as bit count. This implementation will ignore the lower three bits of both values, thus accepting just whole (mapped) bytes as both source address and quantity. Requests with the lower three bits set will be rejected as erroneous.

Rationale: This implementations target processor has neither word operations nor multi-bit shifts. To implement requests not obeying the above restrictions would bring a resource burden not worth the wile.

The same restrictions aplly to WRITE_COILS and READ_DISCRETE_INPUTS. To set a single output bit use WRITE_COIL and to manipulate a selection of bits within a word use MASK_WRITE_REGISTER.

2.15.2.3 #define WRITE_COILS 0x0F

Modbus function code: write bitwise output.

For the interpretation of target address and quantity and the respective restrictions see READ_COILS.

2.15.2.4 #define READ_DISCRETE_INPUTS 0x02

Modbus function code: read bitwise inputs.

For the interpretation of source address and quantity and the respective restrictions see READ_COILS.

2.15.2.5 #define WRITE_READ_REGISTERS 0x17

Modbus function code: write than read word (16 bit) outputs.

This is a time and communication saving combination of WRITE_HOLDING_REGISTERS followed by READ_HOLDING_REGISTERS. This combined output / input usually makes sense if the application maps all (four) Modbus data models to the same address space or uses Modbus' so called holding registers for all analogue, digital or other (process image) input and output.

2.15.3 Typedef Documentation

2.15.3.1 typedef ptfnct_t(funM_t)(struct modThr_data_t ∗thrData)

Type of a protothread function (modThr_data_t ∗)

`funM_t` is a function returning PT_WAITING, PT_YIELDED, PT_EXITED or PT_ENDED and taking a pointer to a Modbus handler (thread) state (modThr_data_t) as parameter.

2.15.3.2 typedef funM_t∗ p2ptFunM

Pointer to a protothread function (modThr_data_t ∗)

`p2ptFunM` is a pointer to a function returning PT_WAITING, PT_YIELDED, PT_EXITED or PT_ENDED and taking a pointer to a Modbus handler (thread) state (modThr_data_t) as parameter.

2.15.4 Function Documentation

2.15.4.1 void modbusInit (void)

Initialise the Modbus (server)

This function initialises the Modbus server. It is (indirectly) called whenever the Ethernet link goes up and hence usually after reset.

At present the implementation does nothing, but that may change in future releases.

2.15.4.2 void registerAppModFun (p2ptFunM *appModFun*)

Register the application Modbus handler function.

Parameters

appModFun	pointer to the function to register; NULL to de-register

See also

modConfData_t

Examples:

main.c.

2.15.4.3 ptfnct_t modbusAppcall (void)

Handle Modbus server events.

This (system) function handles all Modbus events. And it is a as a protothread (part of) the Modbus server state machine. At present this feature is hardly used as this implementation does its work in one step as is recommended for a registered application / user software handler (see modConfData_t.appModFun).

This Modbus server implements the basic communication with the Modbus clients and calls the appropriate registered application software hooks.

2.15.4.4 ptfnct_t appModFun (struct modThr_data_t ∗ *m*)

Handle Modbus server events.

This application function handles the Modbus events prepared and forwarded by the system handler. This function may or may be not implemented by user software. The name could be chosen freely; this declaration is for clarity's sake. If implemented it must be registered prior to incoming Modbus requests to be handled.

For this functions conditions see modConfData_t.appModFun, modConfData, p2ptFunM and funM_t.

Parameters

m	pointer to a Modbus handler (thread) state structure All details of the request to handle are prepared there. The result has to be put there in the appropriate fields respectively via the pointers provided.

Returns

PT_WAITING, PT_YIELDED, PT_EXITED or PT_ENDED
If, as very strongly recommended, all work is done in one stint there's no need for thread organisation and the return value is one of the latter two.

Examples:

main.c.

2.16 Streams

2.16.1 Overview

weAut_01' devices for communication input/output supported by weAutSys can be used directly and via the respective modules / functions. Additionally some of those devices are offered (wrapped) as standard (or stdio.h) streams. In the reset default state stdin/err/out go to or come from the UART.

The reasons for using such streams instead of direct I/O programming / usage are

- prevalent C knowledge and programming habits [conceded]
- generalisation [conceded]

  especially in the sense of
- using stdin/out for providing common tasks and easily switching them between different communication devices [alright]
- using the format string controlled formatting and parsing [don't]

The last point is strongly deprecated. printf, scanf and cognates are quite resource hungry. For a small µController (in a resource saving appliance) and in a non preemptive (Protothreads) runtime this will quickly get a problem and shall be restricted to debugging and may be initialisation before the real (time) work. Using weAutSys' formatting and parsing functions instead will reduce the memory and processor usage by factors five and more.

Files

- file log_streams.h

  Buffered log stream (definitions)
- file streams.h

  Streams (definitions)

Data Structures

- struct streamClassDescript_t

  Extra stream (type) specific data.

Defines

- #define AVAILABLE_UNKNOWN 0xFFFF

  Available input unknown.
- #define LAN_IS_STDOUT

  stdout is the LAN stream output (Ethernet, uIP)
- #define SER_IS_STDOUT

  stdout is the serial output (UART)
- #define USE_AVR_GCC_FILE 1

  use AVR FILE type internals

The buffered log streams

- FILE bufLogStreams

  The stream "device" being a buffer for logging.
- struct thr_data_t ∗ bufLogCons

  The consumer of buffered log streams output (text) data.
- void bufLogStreamsInit (void)

  Initialise the buffered log streams.
- void bufLogStreamsOpen (void)

  Open the buffered log streams.
- uint16_t bufLogInBufferd (FILE ∗streams)

  The number of characters buffered in buffered log streams for input.
- int bufLogGetC (void)

  Get one byte from buffered stream input.
- uint16_t bufLogGetChars (char ∗dst, uint16_t n)

  Get a number of bytes from buffered stream input.
- uint8_t uartPutLogBuf (void)

  Put characters from buffered log streams to serial output.
- uint16_t bufLogStreamsOutSpace (FILE ∗streams) __attribute__((pure))

  The buffer space available for buffered log streams output.
- uint8_t bufLogCheckOutSpace (uint16_t const reqSpace, FILE ∗const streams)

  Check the space available buffered log streams output.
- void bufLogPutC (char const c)

  Put one byte to buffered stream output.
- void bufLogPut2C (char const c1, char const c2)

  Put two bytes to buffered stream output.
- int bufLogGetChar (FILE ∗const stream)

  Get one byte from buffered stream input.
- int bufLogPutChar (char c, FILE ∗const stream)

  Put one byte to buffered stream output.
- int bufLogPutSt (char ∗src, FILE const ∗const stream)

  Put a RAM string (some characters) to buffered stream output.
- int bufLogPutSt_P (char ∗src, FILE const ∗const stream)

  Put some characters from program space to buffered stream output.
- #define BUF_STREAMS_CAP 2047

  The buffered log streams capacity.
- #define bufLogConsIsUART

  The UART is the consumer of buffered log streams output (text) data.
- #define bufLogSetConsumer(tD)

  Set the consumer of buffered log streams output (text) data.
- #define bufLogSetUART()

  Set the UART as consumer of buffered log streams output.
- #define BufLogInBufferd

  The number of characters buffered in buffered log streams for input.

Common stream handling definitions

- typedef uint16_t(∗ str2InBufferd)(const FILE ∗stream)
  Function pointer / type: number of characters buffered for input.
- typedef uint16_t(∗ str2OutSpace)(const FILE ∗stream)
  Function pointer / type: buffer space available for output.
- typedef uint8_t(∗ str2CheckOutSpace)(uint16_t reqSpace, const FILE ∗stream)
  Function pointer / type: checking the space available for output.
- typedef int(∗ str2PutS)(char ∗src, const FILE ∗stream)
  Function pointer / type: writing a RAM string to output.
- typedef int(∗ str2PutS_P)(prog_char ∗src, const FILE ∗stream)
  Function pointer / type: writing a flash memory string to output.
- typedef struct
  streamClassDescript_t streamClassDescript_s
  Extra stream (type) specific data (structure)
- uint16_t inBufferd (const FILE ∗stream)
  Number of characters buffered for input.
- uint16_t outSpace (const FILE ∗stream)
  Buffer space available for output.
- uint8_t checkOutSpace (uint16_t reqSpace, const FILE ∗stream)
  Checking the space available for output.
- int putSt (char ∗src, const FILE ∗stream)
  Write a RAM string to output.
- int putSt_P (char const ∗src, const FILE ∗stream)
  Write a flash memory string to output.
- #define PT_YIELD_OUT_SPACE(pt, requSpace, streams)
  Set standard I/O and exit or yield for requested output space.
- #define PT_WAIT_OUT_SPACE(pt, requSpace, streams)
  Set standard I/O and exit or wait for requested output space.

The serial streams

- FILE serStreams
  The stream "device" connected to the serial IO / UART(0)
- void serStreamsOpen (void)
  Open the serial (UART) streams.
- uint16_t serStreamsOutSpace (FILE ∗streams)
  The buffer space available for serial output.
- uint8_t serCheckOutSpace (uint16_t reqSpace, FILE ∗streams)
  Check the space available for serial output.
- int serPutChar (char c, FILE ∗stream)
  Put one byte to serial output.

Recommended standard I/O functions

- void stdPutC (char c)
  Write a character to `stdout`.
- void logStackS (uint8_t maxBytes)
  Log the stack frame to stdout.
- #define stdPutS(src)
  Write a RAM string to `stdout`.
- #define stdPutS_P(src)
  Write a flash memory string to `stdout`.

The Ethernet / LAN streams

- FILE lanStreams

  The stream "device" connected to the Ethernet (uIP)
- void lanStreamsInit (void)

  Initialise the Ethernet / LAN streams.
- void lanStreamsEcho (void)

  Set the Ethernet / LAN streams to echo input.
- void lanStreamsOpen (void)

  Open the Ethernet / LAN streams.
- void lanStreamSend (void)

  Sends the LAN streams data (over the Ethernet)
- uint16_t lanStreamsOutSpace (FILE ∗streams)

  The buffer space available for Ethernet / LAN streams output.
- uint8_t lanCheckOutSpace (uint16_t reqSpace, FILE ∗streams)

  Check the space available Ethernet / LAN streams output.
- uint16_t lanInBufferd (FILE ∗streams)

  The number of characters buffered from LAN (stream) input.
- int lanGetC (void)

  Get one byte from LAN stream input.
- void lanPutC (char c)

  Put one byte to LAN stream output.
- int lanGetChar (FILE ∗stream)

  Get one byte from LAN stream input.
- int lanPutChar (char c, FILE ∗stream)

  Put one byte to LAN stream output.
- int lanPutChars (char ∗src, uint16_t n)

  Put some characters to LAN stream output.
- int lanPutSt (char ∗src, const FILE ∗stream)

  Put a RAM string (some characters) to LAN stream output.
- int lanPutSt_P (char ∗src, const FILE ∗stream)

  Put some characters from program space to LAN stream output.

The nul device

- FILE nulStreams

  The stream "device" connected to the nul device.
- void nulStreamsOpen (void)

  Open the nul streams.
- uint16_t nulStreamsOutSpace (FILE ∗streams)

  The buffer space available for nul output.
- uint8_t nulCheckOutSpace (uint16_t reqSpace, FILE ∗streams)

  Check the space available for nul streams output.
- uint16_t nullnBufferd (FILE ∗streams)

  The number of characters buffered from nul input.
- int nulGetC (void)

  Get one byte from nul stream input.
- void nulPutC (char c)

  Put one byte to nul stream output.
- int nulPutChar (char c, FILE ∗stream)

  Put one byte to nul stream output.

- int nulGetChar (FILE ∗stream)

  Get one byte from nul stream input.
- int nulPutSt (char ∗const src, const FILE ∗stream)

  Put a string (some characters) to nul stream output.

Functions

- void initStdStreams (FILE ∗initStream)

  Initialise the standard streams.
- void streamsClose (void)

  Close other standard streams.
- uint8_t switchStdStreams (FILE ∗toStream)

  Switch the standard streams to another stream.

2.16.2 Define Documentation

2.16.2.1 #define **BUF_STREAMS_CAP** 2047

The buffered log streams capacity.

The value must be (a power of 2 - 1) and at least 255.

2.16.2.2 #define **bufLogConsIsUART**

The UART is the consumer of buffered log streams output (text) data.

This evaluates to true if the serial output is to consume the buffered logs.

2.16.2.3 #define **bufLogSetConsumer(** *tD* **)**

Set the consumer of buffered log streams output (text) data.

If the consumer is set to `0` or `NULL` there is no consumer and all output will go to the log streams (except the last BUF_STREAMS_CAP bytes will be forgotten).

A value of `1` means the serial interface is the consumer. This presupposes the UART being connected to a (human readable) text mode device.

Any other value must be a pointer to a thread data structure. A suitable thread function has to recognise this and handle the buffered log output. weAutSys' Telnet server e.g. does this.

At any given moment there can be just one or none (0) consumer for the buffered log streams output.

Parameters

tD	pointer to the consumer's thread data structure or 0 or 1

See also

bufLogCons
bufLogConsIsUART
bufLogSetUART

2.16.2.4 #define **bufLogSetUART(** **)**

Set the UART as consumer of buffered log streams output.

This sets the UART as consumer of log output. This presupposes the UART being connected to a (human readable) text mode device.

If this pre-condition is met this may be called by user software's initialisation (phase 2) to prevent the first active Telnet client from automatically get that role. This may be mandatory if Telnet debugging is on.

See also

bufLogCons
bufLogConsIsUART
bufLogSetConsumer

Examples:

main.c.

2.16.2.5 #define **BufLogInBufferd**

The number of characters buffered in buffered log streams for input.

See also

bufLogInBufferd

2.16.2.6 #define **PT_YIELD_OUT_SPACE(** *pt, requSpace, streams* **)**

Set standard I/O and exit or yield for requested output space.

This is a macro to be used inside a protothread.

It sets a re-entry point for the next schedule of `pt` (after eventually yielding or waiting). Then it switches the standard output to `streams` and exits the calling thread `pt` (returning PT_EXITED) if that is not possible (cause `streams` is `NULL` e.g.).

If the switching to / opening of `streams` was feasible this macro then

- does nothing else (just yields once), if the requested space is immediately available, or if the availability can't be determined. Anyhow PT_YIELDED is returned at first entry.

Otherwise the opening of `streams` is un-done and

- yields the protothread `pt`, if the requested space may become available in the future by just waiting
- just blocks the protothread `pt`, if the requested space will not become available by yielding / waiting but may be by other's activities.
- exits and resets the protothread, if the requested space will never become available for the current (set) standard output stream (connection) or if `streams` is not usable or `NULL`

Parameters

pt	the pointer to the protothread control structure.
requSpace	the requested output space
streams	the stream to use and switch stdout to

See also

checkOutSpace()
PT_OR_YIELD_REENTER

Examples:

main.c.

2.16.2.7 #define PT_WAIT_OUT_SPACE(pt, *requSpace*, *streams*)

Set standard I/O and exit or wait for requested output space.

This is a macro to be used inside a protothread.

It sets a re-entry point for the next schedule of `pt` (after eventually yielding or waiting). Then it switches the standard output to `streams` and exits the calling thread `pt` (returning PT_EXITED) if that is not possible (cause `streams` is `NULL` e.g.).

If the switching to / opening of `streams` was feasible this macro then

- does nothing else (just goes ahead), if the requested space is immediately available, or if the availability can't be determined.

Otherwise the opening of `streams` is un-done and

- yields the protothread `pt`, if the requested space may become available in the future by just waiting
- just blocks the protothread `pt`, if the requested space will not become available by yielding / waiting but may be by other's activities.
- exits and resets the protothread, if the requested space will never become available for the current (set) standard output stream (connection) or if `streams` is not usable or `NULL`

Parameters

pt	the pointer to the protothread control structure.
requSpace	the requested output space
streams	the stream to use and switch stdout to

See also

checkOutSpace()
PT_OR_YIELD_REENTER

2.16.2.8 #define SER_IS_STDOUT

stdout is the serial output (UART)

This evaluates to true, if stdout goes to the UART.

2.16.2.9 #define LAN_IS_STDOUT

stdout is the LAN stream output (Ethernet, uIP)

This evaluates to true, if stdout goes via uIP to the Ethernet.

2.16.2.10 #define stdPutS(*src*)

Write a RAM string to `stdout`.

This is an abbreviation for `putSt(src, stdout)`.

Examples:

main.c.

2.16.2.11 #define stdPutS_P(*src*)

Write a flash memory string to `stdout`.

This is an abbreviation for `putSt_P(src, stdout)`.

2.16.3 Typedef Documentation

2.16.3.1 typedef uint16_t(∗ str2InBufferd)(const FILE ∗stream)

Function pointer / type: number of characters buffered for input.

A function of this type returns the number of bytes that can be fetched from the stream's input without getting EOF.

For buffered devices with known state the answer will be correct.

In other cases the number of characters readable in one run without EOF is unknown. Then a large number with bit 15 set (0xFFFF AVAILABLE_UNKNOWN) is returned.

See also

inBuffered

2.16.3.2 typedef uint16_t(∗ str2OutSpace)(const FILE ∗stream)

Function pointer / type: buffer space available for output.

A function of this type returns the number of bytes that can currently be output to the stream in one run without waiting or overflow.

For buffered devices with known state the answer will be correct.

In other cases a large number with bit 15 set (0xFFFF AVAILABLE_UNKNOWN) is returned.

See also

str2CheckOutSpace()
outSpace

2.16.3.3 typedef uint8_t(∗ str2CheckOutSpace)(uint16_t reqSpace, const FILE ∗stream)

Function pointer / type: checking the space available for output.

A function of this type checks the requested space against the buffer space available.

If the stream is, for example the UART, its function checks the requested space against the UART buffer size or available space. Return values are 0, 2 or 4.

If the stream is the LAN respectively the currently handled IP connection the respective function checks the requested space against the actual connection's remaining output space. Return values are 0, 3 or 4.

For other devices with known state the correct / appropriate answer is given.

For an unknown device 1 is returned and for no (NULL) device 4.

Parameters

reqSpace	the number of bytes requested for output in one complete step
stream	the stream In Implementations for one single (singleton) concrete stream, this parameter is usually ignored.

Returns

0: required space is available. I.e. go ahead with planned output.
1: may be available, but can't be determined for the given device / stream. I.e. blindly go ahead or give up;
2: is not available, but may become (automatically in the future, i.e. yielding / waiting is OK;
3: is not available, but may become by own activity (flushing, sending). I.e. just yielding / waiting would be endless. 4: will never be available. I.e. give up or request less space.

See also

PT_YIELD_OUT_SPACE
checkOutSpace

2.16.3.4 typedef int(∗ **str2PutS**)(char ∗src, const FILE ∗stream)

Function pointer / type: writing a RAM string to output.

A function of this type writes the content of the string `src` to the `stream`. It will never block.

It returns the number of characters transferred. If not enough space is available the returned value will be smaller than the length of the string `src`. It may even be 0. In the latter case nothing was output.

Hence this operation should be (indirectly) guarded by outSpace() or PT_YIELD_OUT_SPACE .

The output stops after at the first 0 in src. By that this is a string output function. To send a (binary) 0 use stdPutC() or `fputc()`.

Parameters

src	the characters to be output (pointer to RAM not NULL !)
stream	the destination stream

Returns

the number of characters output

See also

putSt

2.16.3.5 typedef int(∗ **str2PutS_P**)(prog_char ∗src, const FILE ∗stream)

Function pointer / type: writing a flash memory string to output.

A function of this writes the content of the string `src` to the `stream`. It will never block.

It returns the number of characters transferred. If not enough space is available the returned value will be smaller than the length of the string `src`. It may even be 0. In the latter case nothing was output.

Hence this operation should be (indirectly) guarded by outSpace() or PT_YIELD_OUT_SPACE .

The output stops after at the first 0 in src. By that this is a string output function. To send a (binary) 0 use stdPutC() or `fputc()`.

Parameters

src	the characters to be output (pointer to flash memory not NULL !)
stream	the destination stream

Returns

the number of characters output

See also

putSt_P

2.16.4 Function Documentation

2.16.4.1 void **bufLogStreamsInit** (void)

Initialise the buffered log streams.

This function initialises and clears the bufLogStreams.

2.16.4.2 void **bufLogStreamsOpen** (void)

Open the buffered log streams.

This function sets the standard streams stdin/out/err to the bufLogStreams.

This "switching" will be rescinded by streamsClose() .

This function just (re-) opens leaving all else state un-changed.

2.16.4.3 uint16_t **bufLogInBufferd** (FILE ∗ ***streams***)

The number of characters buffered in buffered log streams for input.

This function returns the number of bytes that can be fetched from buffered log streams input by bufLogGetChar(F-ILE∗) respectively bufLogGetC() without getting EOF.

The returned value is in the range 0 .. BUF_STREAMS_CAP. If 0 is returned nothing should be read.

2.16.4.4 int **bufLogGetC** (void)

Get one byte from buffered stream input.

This function never blocks. It returns a byte value read or EOF.

Returns

a character 0..255 or EOF

2.16.4.5 uint16_t **bufLogGetChars** (char ∗ *dst,* uint16_t *n*)

Get a number of bytes from buffered stream input.

This function never blocks. It reads up to `n` bytes from the buffered stream to the buffer `dst`.

This is mainly intended as helper function for consuming and forward the log output buffered so far.

Returns

the number of characters transferred 0 ..`n`

2.16.4.6 uint8_t **uartPutLogBuf** (void)

Put characters from buffered log streams to serial output.

This function will put as much as possible characters from the buffered log streams to the UART. Returned is the number of characters transferred; its type is byte as long as we stick to UART buffers capacity below 256.

Returns

the number of characters transferred

2.16.4.7 uint16_t **bufLogStreamsOutSpace** (FILE ∗ *streams*)

The buffer space available for buffered log streams output.

Returns

0 .. BUF_STREAMS_CAP

Examples:

main.c.

2.16.4.8 uint8_t **bufLogCheckOutSpace** (uint16_t const *reqSpace,* FILE ∗const *streams*)

Check the space available buffered log streams output.

This function might give a positive answer to repeated request for more characters than space available even if that might lead to loosing the oldest output. The rationale for this "lie" is that a standard programmed yielding for output space wont't hold a thread on logging (debug) output more than twice.

Returns

The required space
4: will never be available cause asked for more than BUF_STREAMS_CAP. I.e. give up or request less space;
2: is not available, but may become (automatically) in the future, i.e. yielding / waiting is OK .
1: may be available, but can't be determined. I.e. blindly go ahead or give up. This answer is (wrongly) given on second request instead of 2; the third request will get 0.
0: is available or instead of 2 at third request or with no consumer. I.e. go ahead with planned output.

See also

str2CheckOutSpace

2.16.4.9 void **bufLogPutC** (char const *c*)

Put one byte to buffered stream output.

This function never blocks and does the output. If there was no space in the buffer the "oldest" character there will be overwritten. Hence this operation should be (indirectly) guarded by bufLogStreamsOutSpace().

Parameters

c	the character to be output

2.16.4.10 void **bufLogPut2C** (char const *c1*, char const *c2*)

Put two bytes to buffered stream output.

This function acts like two times bufLogPutC but is a bit more efficient (and readable).

Parameters

c1	the first character to be output
c2	the second character to be output

2.16.4.11 int **bufLogGetChar** (FILE ∗const *stream*)

Get one byte from buffered stream input.

This function never blocks. It returns a byte value read or EOF.

This function may be used directly for input but then bufLogGetC() is a better choice. Usually this function is used indirectly via stdin of stdio.h as this functions signature is what stdio expects of underlying input streams.

Parameters

stream	(passed by stdio input) totally ignored by this function

Returns

a character 0..255 or EOF

2.16.4.12 int **bufLogPutChar** (char *c*, FILE ∗const *stream*)

Put one byte to buffered stream output.

This function does exactly the the same as the preferable bufLogPutC() except for the extra parameter of type FILE ∗ that is for sake of the standard I/O usage.

Parameters

c	the character to be output
stream	(passed by stdio output) totally ignored by this function

Returns

c in case of success, EOF if output buffers full.

2.16.4.13 int **bufLogPutSt** (char ∗ *src,* FILE const ∗const *stream*)

Put a RAM string (some characters) to buffered stream output.

This function never blocks. It returns the number of characters transferred. If not enough buffer space is available the returned value will be smaller than the length of the RAM string `src`. This space restriction applies to the size of the buffer and not to the free space. If the latter is insufficient this function will just overwrite the oldest output.

This operation should be (indirectly) guarded by bufLogStreamsOutSpace() or bufLogCheckOutSpace().

The output stops at the first 0 in `src`. By that this is a string output function. To send a (binary) 0 use bufLogPutC(0).

Parameters

src	the characters to be output
stream	the current stream (not used)

Returns

the number of characters output

2.16.4.14 int **bufLogPutSt_P** (char ∗ *src,* FILE const ∗const *stream*)

Put some characters from program space to buffered stream output.

This function never blocks. It returns the number of characters transferred. If not enough buffer space is available the returned value will be smaller than the length of the flash memory string `src`. This space restriction applies to the size of the buffer and not to the free space. If the latter is insufficient this function will just overwrite the oldest output.

This operation should be (indirectly) guarded by bufLogStreamsOutSpace() or bufLogCheckOutSpace().

The output stops at the first 0 in `src`. By that this is a string output function. To send a (binary) 0 use bufLogPutC(0).

Parameters

src	the characters to be output in flash memory
stream	the current stream (not used)

Returns

the number of characters output

2.16.4.15 uint16_t **inBufferd** (const FILE ∗ *stream*)

Number of characters buffered for input.

This is the common function implementing the standard behavior. It delegates all else cases (late binding) to the stream type specific function if such is set in the stream (type) specific data.

For this function's behavior see the description in its pointer / type.

2.16.4.16 uint16_t **outSpace** (const FILE ∗ *stream*)

Buffer space available for output.

This is the common function implementing the standard behavior. It delegates all else cases (late binding) to the stream type specific function if such is set in the stream (type) specific data.

For this function's behavior see the description in its str2OutSpace()"pointer / type".

2.16.4.17 uint8_t **checkOutSpace** (uint16_t *reqSpace,* const FILE ∗ *stream*)

Checking the space available for output.

This is the common function implementing the standard behavior. It delegates all else cases (late binding) to the stream type specific function if such is set in the stream (type) specific data.

For this function's behavior see the description in its str2CheckOutSpace()"pointer / type".

2.16.4.18 int **putSt** (char ∗ *src,* const FILE ∗ *stream*)

Write a RAM string to output.

This is the common function implementing the standard behavior. It delegates all normal cases (late binding) to the stream type specific function if such is set in the stream (type) specific data; if no such function is set it uses `fputs()`.

For this function's behavior see the description in its pointer / type.

2.16.4.19 int **putSt_P** (char const ∗ *src,* const FILE ∗ *stream*)

Write a flash memory string to output.

This is the common function implementing the standard behavior. It delegates all else cases (late binding) to the stream type specific function if such is set in the stream (type) specific data; if no such function is set it uses `fputs_P()`.

For this function's behavior see the description in its pointer / type.

2.16.4.20 void **initStdStreams** (FILE ∗ *initStream*)

Initialise the standard streams.

This function is usually called once at system start up but might be called any time to change or assure the standard streams.

Anyway it will switch to the (then) actual standard streams acting hence effectively as streamsClose(). (Additionally all extra data for pre-defined streams will be reset / refreshed.) Do not call when these (side-) effects may be disturbing.

If the parameter is NULL and standard streams were set already they are just kept. Otherwise nulStreams are used as default.

Parameters

initStream	pointer to initial stream for standard streams

See also

serStreams
lanStreams
nulStreams

Examples:

main.c.

2.16.4.21 uint8_t **switchStdStreams** (FILE ∗ *toStream*)

Switch the standard streams to another stream.

This function sets the standard streams stdin/out/err to the `toStream`. If `toStream` is NULL nothing is done.

Parameters

toStream	pointer streams to switch the standard streams to

Returns

0: toStream is NULL; 1: stdIO is or was set to toStream

2.16.4.22 void **streamsClose** (void)

Close other standard streams.

This function just reverses the (stream switching) effect of switchStdStreams() by setting back all standard streams to their initial standard source and destination.

2.16.4.23 void **serStreamsOpen** (void)

Open the serial (UART) streams.

This function sets the standard streams stdin/out/err to the serStreams if (and only if) not yet so set.

This "switching" will be rescinded by streamsClose() .

2.16.4.24 uint16_t **serStreamsOutSpace** (FILE ∗ *streams*)

The buffer space available for serial output.

Same as uartOutSpace (other common signature).

2.16.4.25 uint8_t **serCheckOutSpace** (uint16_t *reqSpace,* FILE ∗ *streams*)

Check the space available for serial output.

See also

str2CheckOutSpace

2.16.4.26 int **serPutChar** (char *c,* FILE ∗ *stream*)

Put one byte to serial output.

This function never blocks. It returns the byte `c` output if the operation was feasible or -1 resp. EOF if not. In the latter case `c` is not output.

Hence this operation should be (indirectly) guarded by uartOutSpace().

Parameters

c	the character to be output
stream	unused, irrelevant (signature required by stdio)

Returns

c in case of success, EOF if output buffers full.

2.16.4.27 void **stdPutC** (char *c*)

Write a character to `stdout`.

If USE_AVR_GCC_FILE is defined with a not 0 value this function calls the stdout stream's character output implementation directly.

Without the direct call optimisation the following holds:

If the stdout is the UART or the current LAN this function just calls serPutChar(c, NULL) resp. lanPutC(c) directly.

For all other streams `fputc(c, stdout)` is used.

2.16.4.28 void **logStackS** (uint8_t *maxBytes*)

Log the stack frame to stdout.

Parameters

maxBytes	the max. number of bytes logged including the stack frame

2.16.4.29 void **lanStreamsInit** (void)

Initialise the Ethernet / LAN streams.

This function initialises the lanStreams for usage with uip_appdata newly received or to be sent and additionally opens it.

2.16.4.30 void **lanStreamsEcho** (void)

Set the Ethernet / LAN streams to echo input.

This function should be called after inititialising the lanStreams and before the first output.

The received data will then be set as beginning of the data to be sent. This will be a 1:1 echo if nothing else is output.

2.16.4.31 void **lanStreamsOpen** (void)

Open the Ethernet / LAN streams.

This function sets the standard streams stdin/out/err to the lanStreams.

This "switching" will be rescinded by streamsClose() .

This function just (re-) opens leaving all else state (like buffer indices) un-changed. For a first open use lanStreamsInit() .

2.16.4.32 void **lanStreamSend** (void)

Sends the LAN streams data (over the Ethernet)

This function sends the LAN streams data written so far and closes the LAN streams.

2.16.4.33 uint8_t **lanCheckOutSpace** (uint16_t *reqSpace*, FILE $*$ *streams*)

Check the space available Ethernet / LAN streams output.

See also

str2CheckOutSpace

2.16.4.34 uint16_t lanInBufferd (FILE ∗ *streams*)

The number of characters buffered from LAN (stream) input.

This function returns the number of bytes that can be fetched from LAN (stream) input by lanGetChar(FILE∗) respectively lanGetC() without getting EOF.

The returned value is in the range 0 .. uip_datalen(). If 0 is returned nothing should be read.

2.16.4.35 int lanGetC (void)

Get one byte from LAN stream input.

This function never blocks. It returns a byte value read or EOF.

Hence this operation should be (indirectly) guarded by lanInBufferd().

Returns

a character 0..255 or EOF

2.16.4.36 void lanPutC (char *c*)

Put one byte to LAN stream output.

This function never blocks even if no output operation is feasible. Hence this operation should be (indirectly) guarded by lanStreamsOutSpace().

This function will be used by stdio as output driver entry indirectly by lanPutChar(char c, FILE ∗stream). It may as well be used directly for output to the current IP connection.

Parameters

c	the character to be output

2.16.4.37 int lanGetChar (FILE ∗ *stream*)

Get one byte from LAN stream input.

This function never blocks. It returns a byte value read or EOF.

Hence this operation should be (indirectly) guarded by lanInBufferd().

This function may be used directly for input but then lanGetC(() is a better choice. Usually this function is used indirectly via stdin of stdio.h as this functions signature is what stdio expects of underlying input streams.

Parameters

stream	(passed by stdio input) totally ignored by this function

Returns

a character 0..255 or EOF

2.16.4.38 int **lanPutChar** (char *c*, FILE * *stream*)

Put one byte to LAN stream output.

This function does exactly the the same as the preferable lanPutC() except for the extra parameter of type `FILE *` that is for sake of the standard I/O usage.

Parameters

c	the character to be output
stream	(passed by stdio output) totally ignored by this function

Returns

c in case of success, EOF if output buffers full.

2.16.4.39 int **lanPutChars** (char * *src*, uint16_t *n*)

Put some characters to LAN stream output.

This function never blocks. It returns the number of characters transferred. If not enough space is available the returned value will be smaller than `n` or than the length of the string `src`. It may even be 0. In the latter case nothing was output.

Hence this operation should be (indirectly) guarded by lanStreamsOutSpace().

The output stops after n (or space permitting) characters or at the first 0 in src. By that this is a string output function. To send a (binary) 0 use lanPutC(char c).

Parameters

src	the characters to be output
n	the maximum number of characters to be output

Returns

the number of characters output

2.16.4.40 int **lanPutSt** (char * *src*, const FILE * *stream*)

Put a RAM string (some characters) to LAN stream output.

This function never blocks. It returns the number of characters transferred. If not enough space is available the returned value will be smaller than the length of the string `src`. It may even be 0. In the latter case nothing was output.

Hence this operation should be (indirectly) guarded by lanStreamsOutSpace().

The output stops after at the first 0 in src. By that this is a string output function. To send a (binary) 0 use lanPut-C(char c).

Parameters

src	the characters to be output (0 terminated)
stream	the current Ethernet connection as stream (not used)

Returns

the number of characters output

2.16.4.41 int **lanPutSt_P** (char ∗ *src*, const FILE ∗ *stream*)

Put some characters from program space to LAN stream output.

This function never blocks. It returns the number of characters transferred. If not enough space is available the returned value will be smaller than the length of the string `src`. It may even be 0. In the latter case nothing was output.

Hence this operation should be (indirectly) guarded by lanStreamsOutSpace().

Parameters

src	the characters to be output in flash memory (0 terminated)
stream	the current Ethernet connection as stream (not used)

Returns

the number of characters output

2.16.4.42 void **nulStreamsOpen** (void)

Open the nul streams.

This function sets the standard streams stdin/out/err to the nulStreams.

This "switching" will be rescinded by streamsClose() .

2.16.4.43 uint16_t **nulStreamsOutSpace** (FILE ∗ *streams*)

The buffer space available for nul output.

This function always says 4095. It is provided to implement a stream that swallows and forgets all output.

2.16.4.44 uint8_t **nulCheckOutSpace** (uint16_t *reqSpace*, FILE ∗ *streams*)

Check the space available for nul streams output.

This function always returns 0 i.e. available. It is provided to implement a stream that swallows unlimited amount of output (by just forgetting it).

2.16.4.45 uint16_t **nulInBufferd** (FILE ∗ *streams*)

The number of characters buffered from nul input.

This function returns always 0 as there will be no input from nul. It is provided to implement an input stream that will never deliver anything respectively be always empty.

2.16.4.46 int **nulGetC** (void)

Get one byte from nul stream input.

This function is provided to implement an input stream that will never deliver anything respectively be always empty.

Returns

EOF (always)

2.16.4.47 void **nulPutC** (char *c*)

Put one byte to nul stream output.

This function never blocks and does nothing. It is provided to implement a stream that swallows unlimited amount of output (by just forgetting it).

All other output functions to nul do have no effect, also.

Parameters

c	the character to be output

2.16.4.48 int **nulPutChar** (char *c*, FILE ∗ *stream*)

Put one byte to nul stream output.

This function never blocks. It returns the byte `c` provided as parameter value and does nothing else. It is provided to implement a stream that swallows and forgets all output.

Parameters

c	the character to be output
stream	unused, irrelevant (signature required by stdio)

Returns

c

2.16.4.49 int **nulGetChar** (FILE ∗ *stream*)

Get one byte from nul stream input.

This function returns `EOF`. It is provided to implement an input stream that will never deliver anything respectively always be empty.

Parameters

stream	(passed by stdio input) totally ignored by this function

Returns

`EOF`

2.16.4.50 int **nulPutSt** (char ∗const *src*, const FILE ∗ *stream*)

Put a string (some characters) to nul stream output.

This function never blocks. It returns 1 no matter the parameters and does nothing else. It is provided to implement a stream that swallows unlimited amount of output (by just forgetting it).

As the pointer to the characters to output `src` is not de-referenced this function works for RAM and program (flash) memory strings as well.

Parameters

src	the characters to be output (not used)
stream	(not used)

Returns

1 if src is not null or 0 else

2.16.5 Variable Documentation

2.16.5.1 FILE bufLogStreams

The stream "device" being a buffer for logging.

This stream is a buffer (or pipe or decorator) for output. Its intended use is as buffered log output to humans.

The buffered content is available by input or by bulk copying out of the buffered text to be consumed and send forth by an appropriate device respectively application, be it the UART or one Telnet connection.

Every usage of the log buffered stream device via standard streams shall be embraced by bufLogStreamsOpen() and streamsClose().

Examples:

main.c.

2.16.5.2 struct thr_data_t* bufLogCons

The consumer of buffered log streams output (text) data.

This (16 bit) value points to the state data structure of the application to regularly consume (and send forward) the bufLogStreams text buffered so far.

Special values are:

NULL or 0 : no consuming device or application <br > 1 : the UART shall output all buffered text.

See also

bufLogSetConsumer

2.16.5.3 FILE serStreams

The stream "device" connected to the serial IO / UART(0)

On (default) initialisation this stream will be used for the standard streams.

Examples:

main.c.

2.16.5.4 FILE lanStreams

The stream "device" connected to the Ethernet (uIP)

This stream shall only be used while handling Ethernet messages, i.e. directly or indirectly in uip_appcall(), uip-Appcall() or udp_appcall(). It will work on uip_appdata, i.e. use it as buffer for both directions.

Output functions must not be used before uip_appdata as input (message) is fully processed.

Every usage shall be embraced by lanStreamsOpen() and streamsClose() if used via standard streams.

2.16.5.5 FILE nulStreams

The stream "device" connected to the nul device.

This stream shall only be used as initial standard streams if no other character / byte oriented I/O is permanently available (from reset on). That usually means the UART is reserved for other purposes.

This streams shall be set respectively opened as standard I/O (stdio) whenever no other stream is available therefore. The rationale is any (user) thread might use standard I/O (stdio) any time.

The "device" features all in/output functions optimised for doing nothing. It is always available, i.e. from reset phase 1 on until shutdown.

2.17 Files

2.17.1 Overview

This module supports the adaption of file system implementations (from other sources). weAut_01 has a slot for small memory cards (SMCs). The communication with SMCs via SPI is supported by weAutSys' respective driver implementation.

SMCs are well suited to carry a file system, FAT32 being the quasi standard there. Principally weAut_01' EEPROM and flash memory content could (partly) be organised as file system too.

weAutSys offers file systems for MMCs implemented by ChaN's fatFS. This adaption of fatFS to weAut_01 / weAut-Sys is suitable to handle two constraints

- processor size and clock frequency
- the non preemptive nature of the underlying runtime

User software utilising a SMC file system have, of course, to consider those boundary conditions, too.

Files

- file smc2fs.h

  weAutSys' file system adaption to a small memory card

Data Structures

- struct FS_WORK

  Work space for file system operations (structure FS_WORK)

Defines

- #define fMountSMC()

  Mount / initialise the SMC file system as drive 0.
- #define lockFsWorkFor(ls)

  Set the lock on fsWork.
- #define SMC_FS_CLIUSE 0xCB

  SMC file system (structure) locked for (application) CLI.
- #define SMC_FS_SYSTUSE 0xFB

  SMC file system (structure) locked for runtime / system use.
- #define unlockFsWorkFrom(ls)

  Unset the lock on fsWork.

Functions

- uint8_t stdPutFilInf (FILINFO ∗filInf, const uint8_t inf)

  Write file info data to standard output.

Variables

- FATFS fileSystSMC

  File system object (the one for SMC)
- FS_WORK fsWork

  Work space for file system operations.

2.17.2 Define Documentation

2.17.2.1 #define fMountSMC( )

Mount / initialise the SMC file system as drive 0.

See also: fmount(0, &fileSystSMC);

2.17.2.2 #define SMC_FS_SYSTUSE 0xFB

SMC file system (structure) locked for runtime / system use.

See also

SMC_FS_CLIUSE

2.17.2.3 #define SMC_FS_CLIUSE 0xCB

SMC file system (structure) locked for (application) CLI.

See also

SMC_FS_SYSTUSE

Examples:

main.c.

2.17.2.4 #define lockFsWorkFor(*ls*)

Set the lock on fsWork.

Note

This (macro) is an expression

Parameters

ls	lock signature (uint8_t); see SMC_FS_SYSTUSE

Returns

true if operation was feasible

Examples:

main.c.

2.17.2.5 #define unlockFsWorkFrom(*ls*)

Unset the lock on fsWork.

Note

This (macro) is an expression

Parameters

ls	lock signature (uint8_t) of current lock holder

Returns

false if operation was feasible; true (holder) if other holds lock

Examples:

main.c.

2.17.3 Function Documentation

2.17.3.1 uint8_t stdPutFilInf (FILINFO ∗ *filInf,* const uint8_t *inf*)

Write file info data to standard output.

This function outputs the information on the file `filInf` to the standard output stream.

The bits of the parameter `inf` control the information output:

Bit 7 (0x80) : output leading blank

Bit 5 (0x20) : output last modified date / time

Bit 4 (0x10) : output attributes

Bit 3 (0x08) : output length

Bit 0 (0x01) : output trailing new line

Parameters

filInf	pointer to the file information structure
inf	the informations to output

Returns

0: nothing was output

2.17.4 Variable Documentation

2.17.4.1 FATFS fileSystSMC

File system object (the one for SMC)

This is the (one) file system state object used.

In weAutSys / weAut_01 and alike there can be just one or zero file-system for a small memory card (SMC card inserted.

2.17.4.2 FS_WORK fsWork

Work space for file system operations.

This structure holds the state of the system or application software's ongoing file system operations. In a small realtime system file system operations on SMCs (interfaced via SPI) are relatively expensive in terms of both total processor time and size of indivisible (by yielding) sub-steps.

Hence file system operations should occur in one (well designed / planned) thread at a time only. This structure fsWork of type FS_WORK is to hold all file system related state and should remain singleton for said reasons.

As this variables singleton property is strongly recommended some functions and macros work directly on it. Their services are not available for further instances if application software chooses to have them.

Examples:

main.c.

2.18 + + Utilities and helpers + +

2.18.1 Overview

The helper and utility functions and values being heavily used by weAutSys' internals may and should as well be exploited by application software.

Some of the services are also available by standard C constructs or libraries. The architecture specific implementations here give (partly drastic) improvements in time and resource consumption; that may make their use mandatory in the light of

Modules

- Arithmetic utilities
- Formatters
- Parsers
- Text blocks and utilities
- Persistent storage

Files

- file utils.h

  weAutSys utility / library functions to be used also by application / user software

2.19 Arithmetic utilities

2.19.1 Overview

The arithmetic functions provided by weAutSys fill the gaps of the 8 bit, CISC little endian architecture. Some have no direct substitute.

Those, replacing a C construct or library function are substantially faster. Those modulo, divide and multiply functions are unsigned — even without u in the names &mdash except stated otherwise.

Additionally some constants (values) and logical utilities are provided.

Defines

- #define clearBitMask(bitNum)

  Get the 8 bit "Clear bit mask" for the given bit.
- #define setBitMask(bitNum)

  Get the 8 bit "Set bit mask" for the given bit.

Optimised Divide functions

- u8div_t divByVal10 (uint8_t div) __attribute__((pure))

  Divide an unsigned byte by the constant 10.
- u8div_t divWByVal10toByte (uint16_t div) __attribute__((pure))

  Divide an unsigned word by by the constant 10 for a byte quotient.
- u16div_t divWByVal1000 (uint16_t div) __attribute__((pure))

  Divide an unsigned word by by the constant 1000.
- uint8_t mod16byVal7 (uint16_t val) __attribute__((pure))

  Modulo 7 of an unsigned 16 bit value.
- uint8_t mod8byVal7 (uint8_t val) __attribute__((pure))

  Modulo 7 of an unsigned 8 bit value.
- uint32_t div32byVal512 (const uint32_t div) __attribute__((pure))

  Divide unsigned 32 bit by the constant 512.
- uint16_t div16byVal512 (const uint16_t div) __attribute__((pure))

  Divide unsigned 16 bit by the constant 512.
- uint32_t div32byVal1024 (const uint32_t div) __attribute__((pure))

  Divide unsigned 32 bit by the constant 1024.
- uint32_t div32byVal2048 (const uint32_t div) __attribute__((pure))

  Divide unsigned 32 bit by the constant 2048.
- uint32_t div32byVal256 (const uint32_t div) __attribute__((pure))

  Divide unsigned 32 bit by 256.
- uint32_t div32byVal128 (const uint32_t div) __attribute__((pure))

  Divide unsigned 32 bit by the constant 128.
- #define mod2pow(div, po2)

  Modulo by power of 2.

Optimised Multiplication functions

- uint32_t mul16with8 (const uint16_t f16, const uint8_t f8) __attribute__((pure))

  Multiply unsigned 16 bit with 8 bit.
- uint32_t mul16 (const uint16_t ab, const uint16_t cd) __attribute__((pure))

  Multiply unsigned 16 bit with 16 bit.

- uint32_t mul16with17 (const uint16_t ab, const uint16_t cd) __attribute__((pure))

  Multiply unsigned 16 bit with 17 bit.
- uint32_t mul32withVal512 (const uint32_t fac) __attribute__((pure))

  Multiply unsigned 32 bit with the constant 512.
- uint16_t **mul16withVal512** (const uint16_t fac) __attribute__((pure))

Optimised Compare functions

- uint8_t geU32ModAr (uint32_t a, uint32_t b) __attribute__((always_inline))

  A 32 bit unsigned greater equal (ge) comparison for modulo arithmetic.
- uint8_t leU32ModAr (uint32_t a, uint32_t b) __attribute__((always_inline))

  A 32 bit unsigned less or equal (le) comparison for modulo arithmetic.

Endianess Handling functions

- void toggle32endian (ucnt32_t ∗value)

  Toggle the endianess of a 32 bit value.
- void toggle16endian (ucnt16_t ∗value)

  Toggle the endianess of a 16 bit value.
- uint16_t convert16endian (uint16_t value)

  Convert the endianess of a 16 bit value.
- uint32_t convert32endian (uint32_t value)

  Convert the endianess of a 32 bit value.
- void add16littleTo32bigEndian (ucnt32_t ∗opRes, uint16_t op2)

  Add a normal (little endian) 16 bit value to 32 bit big endian.

Optimised Divide functions (optionally in bootloader)

- uint16_t div16 (uint16_t ∗rem, uint16_t dividend, uint16_t divisor)

  Unsigned 16 bit divide.
- uint32_t div24 (uint32_t ∗rem, uint32_t dividend, uint32_t divisor)

  Unsigned 24 bit divide.
- uint32_t div32 (uint32_t ∗rem, uint32_t dividend, uint32_t divisor)

  Unsigned 32 bit divide.
- uint32_t div32by24 (uint32_t ∗rem, uint32_t dividend, uint32_t divisor)

  Unsigned 32 bit divide by 24 bit.
- uint32_t div32by16 (uint16_t ∗rem, uint32_t dividend, uint16_t divisor)

  Unsigned 32 bit divide by 16 bit.

Variables

- uint8_t const clearBitMasks [8]

  Clear bit masks (table in flash memory)
- uint8_t const setBitMasks [8]

  Set bit masks (table in flash memory)

2.19.2 Define Documentation

2.19.2.1 #define mod2pow(*div,* *po2*)

Modulo by power of 2.

This is a fast modulo implementation for an architecture without a divide instruction. As long as the (AVR-GC) C-compiler does not optimise power of 2 modulo this gives a quite significant performance improvement.

Warning: If `po2` is not an unsigned power of 2 (i.e. just one bit set) this will not give the correct arithmetic result.

Parameters

div	the (unsigned) dividend
po2	the (unsigned) quotient (must be a power of 2 and $>=$ 2)

Returns

div % po2

2.19.2.2 #define setBitMask(*bitNum*)

Get the 8 bit "Set bit mask" for the given bit.

Parameters

bitNum	the bit number; the value will be taken modulo 8, i.e. 0..7

Returns

the mask to set the given bit in a byte (by OR)

See also

setBitMasks
clearBitMask()

Examples:

main.c.

2.19.2.3 #define clearBitMask(*bitNum*)

Get the 8 bit "Clear bit mask" for the given bit.

Parameters

bitNum	the bit number; the value will be taken modulo 8, i.e. 0..7

Returns

the mask to clear the given bit in a byte (by AND)

See also

clearBitMasks
setBitMask()

This array of length 8 in flash memory contains the masks to set bits 0 .. 7 by ORing. Hence the values are 1, 2, 4 .... 128 (0x80).

Examples:

main.c.

2.19.3 Function Documentation

2.19.3.1 u8div_t divByVal10 (uint8_t *div*)

Divide an unsigned byte by the constant 10.

ATmega µ-controllers don't have any divide in their (RISC) instruction set. Nevertheless getting quotient and remainder by 10 in 8 bit unsigned arithmetic is needed quite often (even in loops).

Hence no effort was spared to make this a good implementation for `uint8_t` dividends. This function is faster than direct usage of avr-gcc C compiler's / and % operators and also faster than using stdlib.h's div(...,10). The same is true for all the other divide functions coming with weAutSys and its serial bootloader.

Returns

the quotient and the remainder.

Parameters

div	the dividend (byte, unsigned, 0..255)

See also

divWByVal10toByte(uint16_t)
divWByVal1000(uint16_t)
twoDigs(char∗, uint8_t)

2.19.3.2 u8div_t divWByVal10toByte (uint16_t *div*)

Divide an unsigned word by by the constant 10 for a byte quotient.

ATmega µ-controllers don't have any divide in their (RISC) instruction set. Getting quotient and remainder by 10 is, nevertheless, needed quite often — especially for decimal formatting.

In that use case the dividends (`div`) are often in the range 0..999. That may be achieved for larger numbers by divWByVal1000. Hence this is a highly optimised implementation for those small 16 bit dividends (0..999). It will fail without warning if the `div` is 1286 or above!

It's the user's responsibility not to use this function outside its operation range. On the other hand the implementation is much faster than all equivalents expressed with avr-gcc or libc functions.

Returns

the quotient and the remainder.

Parameters

div	the dividend (16 Bit, unsigned, 0..1286)

See also

divByVal10(uint8_t)
divWByVal1000(uint16_t)

2.19.3.3 u16div_t divWByVal1000 (uint16_t *div*)

Divide an unsigned word by by the constant 1000.

ATmega µ-controllers don't have any divide in their (RISC) instruction set. Getting quotient and remainder by 1000 is sometimes needed — for example as divide and conquer for subsequent divide by 10 on decimal formatting.

This implementation returns quotient and the remainder as 16 bit values (in an u16div_t), even if 8 bit would suffice for the quotient.

Returns

the quotient and the remainder.

Parameters

div	the dividend (16 Bit, unsigned, 0..65535)

See also

divWByVal10toByte(uint16_t)
threeDigs(char∗, uint16_t)

2.19.3.4 uint8_t mod16byVal7 (uint16_t *val*)

Modulo 7 of an unsigned 16 bit value.

This is an efficient result = val % 7 without using any divide functions.

Parameters

val	the unsigned number

Returns

the remainder by 7 for val (0..6)

2.19.3.5 uint8_t mod8byVal7 (uint8_t *val*)

Modulo 7 of an unsigned 8 bit value.

This is an efficient result = val % 7 without using any divide functions.

Parameters

val	the unsigned number

Returns

the remainder by 7 for val (0..6)

2.19.3.6 uint32_t **mul16with8** (const uint16_t *f16*, const uint8_t *f8*)

Multiply unsigned 16 bit with 8 bit.

This function calculates the the product of an 8 and a 16 bit number and returns the (24 bit) result as 32 bit.

The (ASM) implementation is far more efficient and the usage is less error prone as what could be expressed in C and then made by (AVR-) GCC.

Parameters

f16	the 16 bit (unsigned) factor
f8	the 8 bit (unsigned) factor

Returns

the product f16 ∗ f8 (It is a 24 bit value, bits 24..31 are 0)

See also

mul16
mul16with17

2.19.3.7 uint32_t **mul16** (const uint16_t *ab*, const uint16_t *cd*)

Multiply unsigned 16 bit with 16 bit.

This function calculates the the product of two 16 bit numbers and returns the 32 bit result.

The (ASM) implementation is far more efficient and the usage is less error prone as what could be expressed in C and then made by (AVR-) GCC.

Parameters

ab	the first 16 bit (unsigned) factor
cd	the second 16 bit (unsigned) factor

Returns

the product ab ∗ cd

See also

mul16with8
mul16with17

2.19.3.8 uint32_t **mul16with17** (const uint16_t *ab*, const uint16_t *cd*)

Multiply unsigned 16 bit with 17 bit.

This function calculates the the product of a 16 bit number with a 17 bit number and returns the 32 bit result. The second factor `cd` is supplied as 16 bit unsigned number with bit 16 implied as 1. Hence the second factor is in the range 65536 .. 131071, as is e.g SECONDS_IN_DAY.

This implementation is far more efficient as what could be expressed in C and then made by (AVR-) GCC.

Parameters

ab	the first 16 bit (unsigned) factor
cd	the second 17 bit (unsigned) factor

Returns

the product ab ∗ (cd + 0x10000)

See also

mul16with8
mul16

2.19.3.9 uint32_t **mul32withVal512** (const uint32_t *fac*)

Multiply unsigned 32 bit with the constant 512.

This function calculates the the product of a 32 bit number with the constant 512 and returns the 32 bit result. This multiplication is needed for certain memory card operations (sector number to byte address e.g.).

This implementation is far more efficient as what could be expressed in C and then made by (AVR-) GCC.

Parameters

fac	the 32 bit (unsigned) factor fac's upper 9 bits are ignored and should be 0 to get a arithmetically correct result

Returns

the product fac ∗ 512

See also

mul16with8
mul16
div32byVal512

2.19.3.10 uint32_t **div32byVal512** (const uint32_t *div*)

Divide unsigned 32 bit by the constant 512.

This function calculates the the quotient of a 32 bit number resulting from division by 512 and returns the 32 bit result. This division is needed for certain memory card operations (with 512 byte sectors e.g.).

This implementation is far more efficient as what could be expressed in C and then made by (AVR-) GCC.

Parameters

div	the 32 bit (unsigned) dividend

Returns

the quotient div / 512

See also

mul32withVal512
div16byVal512
div32byVal1024
div32byVal2048

2.19.3.11 uint16_t **div16byVal512** (const uint16_t *div*)

Divide unsigned 16 bit by the constant 512.

This function calculates the the quotient of a 16 bit number resulting from division by 512 and returns the 16 bit result. This division is needed for certain memory card operations (with 512 byte sectors e.g.).

This implementation is far more efficient as what could be expressed in C and then made by (AVR-) GCC.

Parameters

div	the 16 bit (unsigned) dividend

Returns

the quotient div / 512

See also

div32byVal512
mul32withVal512
div32byVal1024
div32byVal2048

2.19.3.12 uint32_t **div32byVal1024** (const uint32_t *div*)

Divide unsigned 32 bit by the constant 1024.

This function calculates the the quotient of a 32 bit number resulting from division by 1024 and returns the 32 bit result. This division is needed for certain memory card operations.

This implementation is far more efficient as what could be expressed in C and then made by (AVR-) GCC.

Parameters

div	the 32 bit (unsigned) dividend

Returns

the quotient div / 1024

See also

mul32withVal512

2.19.3.13 uint32_t **div32byVal2048** (const uint32_t *div*)

Divide unsigned 32 bit by the constant 2048.

This function calculates the the quotient of a 32 bit number resulting from division by 2048 and returns the 32 bit result. This division is needed for certain memory card operations.

This implementation is far more efficient as what could be expressed in C and then made by (AVR-) GCC.

Parameters

div	the 32 bit (unsigned) dividend

Returns

the quotient div / 2048

See also

mul32withVal512

2.19.3.14 uint32_t div32byVal256 (const uint32_t *div*)

Divide unsigned 32 bit by 256.

This function calculates the the quotient of a 32 bit number resulting from division by 2048 and returns the 32 bit result. This division is needed for certain memory card operations.

This implementation is far more efficient as what could be expressed in C and then made by (AVR-) GCC.

Parameters

div	the 32 bit (unsigned) dividend

Returns

the quotient div / 256

See also

mul32withVal512

2.19.3.15 uint32_t div32byVal128 (const uint32_t *div*)

Divide unsigned 32 bit by the constant 128.

This function calculates the the quotient of a 32 bit number resulting from division by 2048 and returns the 32 bit result. This division is needed for certain memory card operations.

This implementation is far more efficient as what could be expressed in C and then made by (AVR-) GCC.

Parameters

div	the 32 bit (unsigned) dividend

Returns

the quotient div / 128

See also

mul32withVal512

2.19.3.16 uint8_t geU32ModAr (uint32_t *a*, uint32_t *b*)

A 32 bit unsigned greater equal (ge) comparison for modulo arithmetic.

This function will return true if a $>=$ b, whereby a and b are considered as (counter) values in modulo 2$**$32 arithmetic. In that sense geU32ModAr(0x0001, 0xFFFE) will return true as 0x0001 is (as counter / modulo value) by 2 higher than 0xFFFE.

That will work correctly for differences less than 2∗∗31 and, as said, very well over the "wrapping border".

Note

Those who remember a Windows98 exploding after running a bit longer than 49 days will know why we need this type of comparison for a 32 bit milliseconds counter (e.g.).

Parameters

a	will be compared to
b	by a >= b in modulo arithmetic sense

See also

msClock()
leU32ModAr

2.19.3.17 uint8_t **leU32ModAr** (uint32_t *a*, uint32_t *b*)

A 32 bit unsigned less or equal (le) comparison for modulo arithmetic.

This function will return true if a <= b, whereby a and b a considered as (counter) values in modulo 2∗∗32 arithmetic. It is the counterpart to geU32ModAr() (but, n.b., not the inverse).

Parameters

a	will be compared to
b	by a <= b in modulo arithmetic sense

See also

geU32ModAr

2.19.3.18 void **toggle32endian** (**ucnt32_t** ∗ *value*)

Toggle the endianess of a 32 bit value.

Parameters

value	the value to convert

See also

toggle16endian(ucnt16_t ∗)

2.19.3.19 void **toggle16endian** (**ucnt16_t** ∗ *value*)

Toggle the endianess of a 16 bit value.

Parameters

value	the value to convert

See also

toggle32endian(ucnt32_t ∗)
convert16endian

2.19.3.20 uint16_t **convert16endian** (uint16_t *value*)

Convert the endianess of a 16 bit value.

Parameters

value	the value to convert

Returns

value with inverted endianess

See also

toggle6endian

2.19.3.21 uint32_t **convert32endian** (uint32_t *value*)

Convert the endianess of a 32 bit value.

Parameters

value	the value to convert

Returns

value with inverted endianess

See also

toggle6endian

2.19.3.22 void **add16littleTo32bigEndian** (ucnt32_t ∗ *opRes,* uint16_t *op2*)

Add a normal (little endian) 16 bit value to 32 bit big endian.

This type of addition is needed by uIP and this is the "good" architecture specific implementation.

Parameters

opRes	the 32 bit big endian (network order) sum and summand
op2	the 16 bit little endian summand

See also

toggle32endian(ucnt32_t ∗)

2.19.3.23 uint16_t **div16** (uint16_t * *rem*, uint16_t *dividend*, uint16_t *divisor*)

Unsigned 16 bit divide.

This functions's effect is

```
result (return)  = dividend / divisor;
*rem  = dividend % divisor;
```

As of June 13 this function is more than twice as fast — and shorter — as the code generated by the AVR-GCC C compiler.

Anyway divide is a quite expensive operation for an 8 bit RISC machine without a divide instruction. Hence the using divide operators (and their direct replacements e.g. by this function, even though optimised as said) is discouraged. For special cases use the faster implementations like divWByVal1000() — or, at least, use the function with the smallest possible operand and result types.

Parameters

rem	pointer to remainder; if null the remainder is lost
dividend	the number to be divided
divisor	the divisor

Returns

the quotient `divisor` / `dividend`

See also

div24
div32

2.19.3.24 uint32_t **div24** (uint32_t * *rem*, uint32_t *dividend*, uint32_t *divisor*)

Unsigned 24 bit divide.

This functions's effect is

```
result (return)  = dividend / divisor;
*rem  = dividend % divisor;
```

This function is quite similar to div32. It also shares the 32 bit parameter and return types (as avr-gcc C has no 24 bit type). Nevertheless it just operates on and returns 24 bit unsigned numbers — just ignoring respectively setting 0 the upper byte.

So it is about twice as fast as the optimised div32.

Parameters

rem	pointer to the 32 bit variable for the 24 bit remainder; if null the remainder is lost; upper byte will be 0.
dividend	the 24 bit number to be divided (upper byte ignored)
divisor	the 24 bit divisor (upper byte ignored)

Returns

the 24 bit quotient `divisor` / `dividend` (upper byte 0)

See also

div16

2.19.3.25 uint32_t div32 (uint32_t * *rem*, uint32_t *dividend*, uint32_t *divisor*)

Unsigned 32 bit divide.

This functions's effect is

```
result (return)  = dividend / divisor;
*rem  = dividend % divisor;
```

As of June 13 this function is more than twice as fast — and shorter — as the code generated by the AVR-GCC C compiler.

32 Bit divide is a very expensive operation for an 8 bit RISC machine without a divide instruction and only single bit shifts for one byte.

Hence the using it this division by C operators and % or even this optimimised replacement function is discouraged. For special cases use the faster implementations like divWByVal1000() — or, at least, use the function with the smallest possible operand and result types. div16 e.g. will be four times faster.

Parameters

rem	pointer to remainder; if null the remainder is lost
dividend	the number to be divided
divisor	the divisor

Returns

the quotient `divisor` / `dividend`

See also

div16
div24

2.19.3.26 uint32_t div32by24 (uint32_t * *rem*, uint32_t *dividend*, uint32_t *divisor*)

Unsigned 32 bit divide by 24 bit.

This functions's effect is

```
result (return)  = dividend / divisor;
*rem  = dividend % divisor;
```

This function is quite similar to div32 respectively div24 or a mix of both. It shares the 32 bit parameter and return types (as avr-gcc C has no 24 bit type).

The divisor and remainder are 24 bit numbers (upper 32 bit byte ignored or set 0). The dividend and the quotient are 32 bit.

It is about 20% faster than div32 (which is quite a lot on a RICS without division).

Parameters

rem	pointer to the 32 bit variable for the 24 bit remainder; if null the remainder is lost; upper byte will be 0.
dividend	the 32 bit number to be divided
divisor	the 24 bit divisor (upper 32 bit byte ignored)

Returns

the 32 bit quotient `divisor` / `dividend` (upper byte 0)

See also

div16

2.19.3.27 uint32_t **div32by16** (uint16_t * *rem*, uint32_t *dividend*, uint16_t *divisor*)

Unsigned 32 bit divide by 16 bit.

This functions's effect is

```
result (return)  = dividend / divisor;
*rem  = dividend % divisor;
```

This function is quite similar to div32 respectively div16 or a mix of both.

The divisor and remainder are 16 bit numbers. The dividend and the quotient are 32 bit.

It is about 35% faster than div32 (which is quite a lot on a RICS without division).

Parameters

rem	pointer to the 16 bit remainder; if null the remainder is lost
dividend	the 32 bit number to be divided
divisor	the 16 bit divisor

Returns

the 32 bit quotient `divisor` / `dividend`

See also

div24

2.19.4 Variable Documentation

2.19.4.1 uint8_t const **setBitMasks**[8]

Set bit masks (table in flash memory)

This array of length 8 in flash memory contains the masks to set bits 0 .. 7 by ORing. Hence the values are 1, 2, 4 .... 128 (0x80).

See also

clearBitMasks
setBitMask()

2.19.4.2 uint8_t const **clearBitMasks**[8]

Clear bit masks (table in flash memory)

This array of length 8 in flash memory contains the masks to clear bits 0 .. 7 by ANDing. Hence the values are ∼1, ∼2, ∼4 .... ∼128 (0x7F).

See also

setBitMasks
clearBitMask()

2.20 Formatters

2.20.1 Overview

The formatter function provided by weAutSys are a (quite) faster alternative to using stdio format strings.

Formatting either return void (a) or a pointer (b) to the character after the last one set by the documented formating process.

In the void case (a) the formatting functions are changing a fixed and documented number of characters in the destination string respectively character array.

In the return pointer case (b) the (next) character position pointed to will have been set 0. This automatically provides an end of string if not overridden by using the returned pointer. Anyway in cases (b) the caller has to provide space for this trailing zero.

Functions

- void eightHexs (char ∗begin, uint32_t value)

  Set an eight digit hexadecimal number with leading zeroes into a string.
- void eightHexsBE (char ∗begin, uint32_t value)

  Set an eight digit big endian hexadecimal number with into a string.
- char ∗ form16BytSeq (char ∗s, uint8_t ∗seq)

  Format a sequence of 16 bytes in four hex groups and as characters.
- char ∗ formDateIso (char ∗s, date_t ∗tDat)

  Format a date (date_t structure)
- char ∗ formFATdate (char ∗s, uint16_t fatDate)

  Format a FAT date.
- char ∗ formFATtime (char ∗s, uint16_t fatTime)

  Format a FAT time.
- char ∗ formIpAdd (char ∗s, uip_ipaddr_t ipAddr)

  Format an ipV4 Ethernet address.
- char ∗ formIpConf (char ∗s, ipConf_t ∗ipConf)

  Format an IP configuration (w/o DHCP)
- char ∗ formIpConfDHCP (char ∗s, ipConf_t ∗ipConf)

  Format an IP configuration, the DHCP part.
- char ∗ formMacAdd (char ∗s, eth_addr_t ∗mac)

  Format a MAC address.
- char ∗ formModTelegr (char ∗s, struct modTelegr_t ∗modTeleg)

  Format a Modbus telegram.
- char ∗ formN20BytSeq (char ∗s, uint8_t ∗seq, uint8_t n)

  Format a sequence of 0..20 bytes in five hex groups and as characters.
- char ∗ formTimDur (char ∗s, datdur_t ∗tRun)

  Format a time (datdur_t structure) as duration.
- char ∗ formTimOfD (char ∗s, datdur_t ∗tTim)

  Format a time (datdur_t structure) as time of day.
- char ∗ formWdShort (char ∗s, uint8_t wd)

  Format the day of week as short clear text.
- void fourDigs (char ∗begin, uint16_t value)

  Set a four decimal digit number into a string.
- void fourHexs (char ∗begin, uint16_t value)

  Set a four digit hexadecimal number with leading zeroes into a string.
- void fourHexsBE (char ∗begin, uint16_t value)

Set an four digit big endian hexadecimal big endian number into a string.

- void threeDigs (char ∗begin, uint16_t value)

  Set a three decimal digit number with leading zeroes into a string.
- void threeDigsB (char ∗begin, uint8_t value)

  Set a three decimal digit number with leading zeroes into a string.
- void twoDigs (char ∗begin, uint8_t value)

  Set a two decimal digit number with leading zeroes into a string.
- void twoHexs (char ∗s, uint8_t value) __attribute__((always_inline))

  Set a two hexadecimal digit number with leading zeroes into a string.

Variables

- char ∗ wDaysShort [ ]

  flash array of the (flash) short weekdays

2.20.2 Function Documentation

2.20.2.1 char∗ **formMacAdd** (char ∗ *s,* **eth_addr_t** ∗ *mac*)

Format a MAC address.

The MAC address from the structure `mac` will be formatted as 11:22:33:44:55:66 delivering 18 characters in `s` including a trailing zero.

Parameters

s	start position within string respectively char array
mac	the (6 byte) address structure

Returns

pointer to the (0-) character following the formatted MAC-address

See also

formIpAdd
formIpConf

2.20.2.2 char∗ **formIpAdd** (char ∗ *s,* **uip_ipaddr_t** *ipAddr*)

Format an ipV4 Ethernet address.

The IP address from the structure `ipAddr` supplied will be formatted as 112.12.0.1 delivering 8..16 characters including a trailing zero.

Parameters

s	start position within string respectively char array (length 16)
ipAddr	the (uint16-t[2] big endian) address structure

Returns

pointer to the (0-) character following the formatted address

See also

formMacAdd
formIpConf

2.20.2.3 char∗ formIpConf (char ∗ *s*, ipConf_t ∗ *ipConf*)

Format an IP configuration (w/o DHCP)

This function formats the IP configuration `ipConf` to a multi-line text.

The length of the text delivered (to `*s`) is up to 165 characters including a trailing zero.

The output does not contain the DHCP information.

Parameters

s	start position within string respectively char array
ipConf	pointer to the IP configuration (addresses, masks, flags)

Returns

pointer to the (0-) character following the formatted configuration

See also

formMacAdd
formIPAdd
formIpConfDHCP

2.20.2.4 char∗ formIpConfDHCP (char ∗ *s*, ipConf_t ∗ *ipConf*)

Format an IP configuration, the DHCP part.

This function formats the IP configuration to a multi-line text.

The length of the text delivered (to `*s`) is up to 65 characters including a trailing zero.

The output does only contain the DHCP information.

Parameters

s	start position within string respectively char array
ipConf	pointer to the IP configuration (addresses, masks, flags)

Returns

pointer to the (0-) character following the formatted configuration

See also

formMacAdd
formIPAdd
formIpConf

2.20.2.5 char∗ formFATtime (char ∗ *s*, uint16_t *fatTime*)

Format a FAT time.

This function formats a 16 bit time delivered as (old fashined) FAT time structure.

The format will be

```
"23:59:58"
```

.

The length of the text delivered (to `*s`) is 9 characters including a trailing zero.

Note that the seconds in that (old DOS) FAT time format are 2 s increments so no odd value for seconds is possible.

Parameters

s	start position within string resp. char array
fatTime	the 16 bit FAT time structure

Returns

pointer to the (0-) character following the formatted time

See also

getFATtime()

2.20.2.6 char* **formFATdate** (char * *s*, uint16_t *fatDate*)

Format a FAT date.

This function formats a 16 bit delivered as (old fashined) FAT date structure (upper 16 bits 16..31).

The format (ISO 8601 / EN 28601) will be

```
"2009-01-31"
```

.

The length of the text delivered (to `*s`) is 11 characters including a trailing zero.

Parameters

s	start position within string resp. char array
fatDate	the 16 bit FAT date structure

Returns

pointer to the (0-) character following the formatted date

See also

getFATtime()

2.20.2.7 char* **formTimDur** (char * *s*, **datdur_t** * *tRun*)

Format a time (datdur_t structure) as duration.

This function formats a time structure datdur_t as duration.

The format will be

```
"1234d 23h59m59s"
```

. 16 characters will be put to `*s` including a trailing zero.

Parameters

s	start position within string resp. char array (not NULL!)
tRun	pointer to the time structure (not NULL!)

Returns

points to the (0-) character behind the formated duration

2.20.2.8 char* formTimOfD (char * s, datdur_t * tTim)

Format a time (datdur_t structure) as time of day.

This function formats a time structure datdur_t as time of day or clock time.
The format will be

```
"23:59:59"
```

. 9 characters will be put to `*s` including a trailing zero.

The field `d` of `tTim` is not used for the result.

Parameters

s	start position within string resp. char array
tTim	pointer to the time structure

Returns

points to the (0-) character behind the formated duration

2.20.2.9 char* formWdShort (char * s, uint8_t wd)

Format the day of week as short clear text.

This function puts the three character short English day of week into `s`.
The format for `wd` 1..7 will be

```
"Sun" "Mon" "Tue" "Wed" "Thu" "Fri" "Sat"
```

.

4 characters will be put `*s` including a trailing zero.

For `wd` outside 1..7 just `*s` will be set 0 and `s` is returned.

Parameters

s	start position within string resp. char array
wd	1..7 is Sunday .. Saturday

Returns

points to the (0-) character behind the formated day of week

2.20.2.10 char∗ **formDateIso** (char ∗ *s,* date_t ∗ *tDat*)

Format a date (date_t structure)

This function formats a date (structure date_t).
The format (ISO 8601 / EN 28601) will be

```
"2009-01-31"
```

.

11 characters will be put `*s` including a trailing zero.

The field `wd` is not used for the result.

Parameters

s	start position within string resp. char array
tDat	pointer to the time structure

Returns

points to the (0-) character behind the formated day of week

2.20.2.11 void **threeDigs** (char ∗ *begin,* uint16_t *value*)

Set a three decimal digit number with leading zeroes into a string.

This function converts a number 0..999 into three characters "000" .. "999". Values of 1000 and above will get "∗∗∗".

Parameters

begin	start position within the target string resp. char array
value	the value to convert

See also

twoDigs(char∗, uint8_t)
fourDigs(char∗, uint8_t)
divWByVal1000(uint16_t)
divWByVal10toByte(uint16_t)
threeDigsB

2.20.2.12 void **threeDigsB** (char ∗ *begin,* uint8_t *value*)

Set a three decimal digit number with leading zeroes into a string.

This function converts an eight bit number 0..255 into three characters "000" .. "255".

Parameters

begin	start position within the target string resp. char array
value	the value to convert

See also

twoDigs(char∗, uint8_t)
fourDigs(char∗, uint8_t)
divWByVal1000(uint16_t)
divWByVal10toByte(uint16_t)
threeDigs

Examples:

main.c.

2.20.2.13 void fourDigs (char ∗ *begin,* uint16_t *value*)

Set a four decimal digit number into a string.

This function converts a number 0..9999 into three characters " 0" .. "9999". Values of 10000 and above will get " ∗∗0".

Parameters

begin	start position within the target string resp. char array
value	the value to convert

See also

twoDigs(char∗, uint8_t)
threeDigs(char∗, uint8_t)
fourHexs((char∗, uint16_t)
divWByVal1000(uint16_t)
divWByVal10toByte(uint16_t)

2.20.2.14 void twoDigs (char ∗ *begin,* uint8_t *value*)

Set a two decimal digit number with leading zeroes into a string.

This function converts a number 0..99 into two characters "00" .. "99". Values of 100 and above will get "∗∗".

Parameters

begin	start position within the target string resp. char array
value	the value to convert

See also

threeDigs(char∗, uint16_t)
divByVal10(uint8_t)

2.20.2.15 void twoHexs (char ∗ *s,* uint8_t *value*)

Set a two hexadecimal digit number with leading zeroes into a string.

This function converts a number 0..255 into two characters "00" .. "FF".

Parameters

s	start position within the target string resp. char array
value	the value to convert

See also

twoDigs(char∗, uint8_t)
threeDigs(char∗, uint16_t)
fourHexs((char∗, uint16_t)
divByVal10(uint8_t)

Examples:

main.c.

2.20.2.16 void **fourHexs** (char ∗ *begin,* uint16_t *value*)

Set a four digit hexadecimal number with leading zeroes into a string.

This function converts a 16 bit number into four characters "0000" .. "FFFF".

Parameters

begin	start position within the target string resp. char array
value	the value to convert

See also

twoDigs(char∗, uint8_t)
twoHexs(char∗, uint8_t)
threeDigs(char∗, uint16_t)
eightHexs(char ∗, uint32_t)
divByVal10(uint8_t)

Examples:

main.c.

2.20.2.17 void **fourHexsBE** (char ∗ *begin,* uint16_t *value*)

Set an four digit big endian hexadecimal big endian number into a string.

This function does the same as eightHexs except for the wrong andianess of the `value` parameter.

Parameters

begin	start position within the target string resp. char array
value	the big endian value to convert

2.20.2.18 void **eightHexs** (char ∗ *begin,* uint32_t *value*)

Set an eight digit hexadecimal number with leading zeroes into a string.

This function converts a 32 bit number into eight characters "00000000" .. "FFFFFFFF".

Parameters

begin	start position within the target string resp. char array
value	the value to convert

See also

twoDigs(char∗, uint8_t)
twoHexs(char∗, uint8_t)
threeDigs(char∗, uint16_t)
fourHexs(char ∗, uint16_t)
eightHexsBE(char ∗, uint32_t)

2.20.2.19 void **eightHexsBE** (char ∗ *begin*, uint32_t *value*)

Set an eight digit big endian hexadecimal number with into a string.

This function does the same as eightHexs except for the wrong andianess of the `value` parameter.

Parameters

begin	start position within the target string resp. char array
value	the big endian value to convert

2.20.2.20 char∗ **form16BytSeq** (char ∗ *s*, uint8_t ∗ *seq*)

Format a sequence of 16 bytes in four hex groups and as characters.

This function formats the first 16 bytes pointed to by `seq` in four groups of 8 hexadecimal digits separated by space followed by said 16 bytes as 16 character text. There control characters (0..0x20 and 0x80..0xA0) are put as blank space.

The result's length is 55 characters including a trailing blank and terminating 0.

The result looks like `20746578 74206F6E 20612032 4720534D text on a 2G SM`

Parameters

s	start position within string respectively char array
seq	pointer to the 16 byte sequence to be formatted

Returns

points to the (0-) character behind the formated result

See also

formN20BytSeq

2.20.2.21 char∗ **formN20BytSeq** (char ∗ *s*, uint8_t ∗ *seq*, uint8_t *n*)

Format a sequence of 0..20 bytes in five hex groups and as characters.

This function formats the `c` (max. 20) bytes pointed to by `seq` in five groups of 8 hexadecimal digits separated by dot (.) followed by 20 bytes as 20 character text. In this text field control characters (0..0x20 and 0x80..0xA0) as well as the bytes [>= `n`] are put as blank space.

The maximum length of the result (for `n=20`) is 68 characters including a trailing blank and terminating 0.

The result (for `n=17`) looks like `20746578.74206F6E.20612032.4720534D.43....... text on a 2G SMC`

Parameters

s	start position within string resp. char array
seq	pointer to the up to 20 byte sequence to be formatted
n	0..20 number of bytes

Returns

points to the (0-) character behind the formated result

See also

form16BytSeq

2.20.2.22 char∗ formModTelegr (char ∗ *s*, struct modTelegr_t ∗ *modTeleg*)

Format a Modbus telegram.

This function formats a Modbus TCP/IP telegram provided as structure `modTeleg`.. This consists of seven bytes called MBAB header plus 11..13 bytes called PDU. From the MBAB the sequence number (#), bytes to follow (+) and the unit (u) are formatted. The PDU part starts with the function code (fx) followed by 0..6 words (16 bit) The number and interpretation of the latter is dependent of the function code and direction (request / response).

The format returned is

" #.... +.... u.. fx.. .... "

All values are formated hexadecimal; 16 bit numbers are interpreted as (wrong) big endian.

Parameters

s	start position within string resp. char array
modTeleg	pointer to the Modbus TCP/IP telegram

Returns

points to the (0-) character behind the formated result

2.21 Parsers

2.21.1 Overview

weAutSys's parser functions provide a fast and dividable (by thread yield) way to handle command line and other text inputs.

Standard signature for parsing functions is

```
uint8_t parseResult(resultType * result, char * s, uint8_t whereParseStarts)
```

return is the index of the character in s not used for getting the result respectively next character index. In case of no success 0 is returned and result will be unchanged.

Note: It is considered to change that signature to

```
uint8_t parseResult(resultType * result, char * whereParseStarts)
```

return is the number of characters consumed for getting the result. In case of no success 0 is returned and result will be unchanged.

The change considered for future would allow some performance gain for the parsing, but slightly complicate the handling.

Defines

- #define INDEX_OFFSET_LIST2 125

  Offset for token found in second list by searchTokenIn.

Functions

- uint8_t asDigit (uint8_t c) __attribute__((always_inline))

  Recognise one digit.
- char ∗ getFirstSVNtokenP (char ∗dest, char const ∗src, uint8_t mxLen)

  Copy the first SVN token or a full string from program space.
- uint8_t isContainedIn (char s[ ], const uint8_t c)

  Check if a character is contained in a RAM string.
- uint8_t isContainedIn_P (char s[ ], const uint8_t c)

  Check if a character is contained in a flash string.
- uint8_t parse2hex (uint8_t ∗res, char s[ ], uint8_t si)

  Parse a two digit hexadecimal number.
- uint8_t parseByteNum (uint8_t ∗res, char s[ ], uint8_t si)

  Parse a one byte number.
- uint8_t parseDate (date_t ∗dat, char s[ ], uint8_t si)

  Parse a date.
- uint8_t parseDWordNum (uint32_t ∗res, char s[ ], uint8_t si)

  Parse a four byte number.
- uint8_t parseIpAdd (uip_ipaddr_t ipAddr, char s[ ], uint8_t si)

  Parse an ipV4 Ethernet address.
- uint8_t parseMACadd (eth_addr_t ∗macAddr, char s[ ], uint8_t si)

  Parse a MAC address.
- uint8_t parseTim (datdur_t ∗tim, char s[ ], uint8_t si)

  Parse a (clock) time.
- uint8_t parseWordNum (uint16_t ∗res, char s[ ], uint8_t si)

Parse a two byte number.
- uint8_t searchFirstToken (char s[ ], uint8_t si, uint8_t len)

  Search the first token in a short (RAM) string.
- uint8_t searchTokenEnd (char s[ ], uint8_t si, uint8_t len)

  Search the the end of a token in a short (RAM) string.
- uint8_t searchTokenIn (char s[ ], uint8_t si, uint8_t se, char const ∗const ∗list, char const ∗const ∗list2)

  Match a (short) token in a (RAM) string to a list of flash strings.
- uint8_t searchTokenStart (char s[ ], uint8_t si, uint8_t len)

  Search the next token in a short (RAM) string.

2.21.2 Function Documentation

2.21.2.1 uint8_t parseIpAdd (uip_ipaddr_t *ipAddr*, char *s[ ]*, uint8_t *si*)

Parse an ipV4 Ethernet address.

This function tries to interpret a part of the string s starting at `si` as an IP address expected in the form "112.12.0.1" i.e. four dot separated numbers of up to three decimal digits each.

Leading blanks will be skipped.

Parameters

ipAddr	pointer to the result (ipaddr_t is an array type); the result is changed only if an IP address could be parsed successfully
s	the RAM string containing the digit(s)
si	the first digit's index in s

Returns

the index of the first character after the parsed IP address in case of success or 0 else

2.21.2.2 uint8_t parseMACadd (eth_addr_t ∗ *macAddr*, char *s[ ]*, uint8_t *si*)

Parse a MAC address.

This function tries to interpret a part of the string s starting at `si` as a MAC address expected in the form "11:2B:3-C:4D:5E:6F" i.e. six colon separated hexadecimal numbers of (up to) two digits each.

Leading blanks will be skipped.

Parameters

macAddr	pointer to the result (eth_addr_t is a structure containing an array type); the result is changed only if a MAC address could be parsed successfully
s	the RAM string containing the digit(s)
si	the first digit's index in s

Returns

the index of the first character after the parsed IP address in case of success or 0 else

2.21.2.3 uint8_t parseTim (datdur_t ∗ *tim*, char *s[ ]*, uint8_t *si*)

Parse a (clock) time.

This function tries to interpret a part of the string `s` starting at `si` as a (clock) time. The expected format is "hh:mm-:ss" or "hh:mm" with the seconds implied as 0. Returned is the index of the first character after that (parsed) time in case of success and 0 otherwise.

Leading and trailing blanks at every segment will be skipped as will be (allowed) leading zeroes. Hence "17:3:9" and " 17 : 03 :09 " will get the same interpretation.

Parameters

tim	pointer to the result structure; may be changed partially even if the parse was not successful
s	the RAM string containing the time
si	the first time's index in s

Returns

the index of the first character after the parsed time in case of success or 0 else

2.21.2.4 uint8_t parseDate (date_t * *dat,* char *s[],* uint8_t *si*)

Parse a date.

This function tries to interpret a part of the string `s` starting at `si` as a (clock) time. The expected format is "yyyy-mm-dd" or "yy-mm-dd" with two digit year interpreted as 2008 .. 2107 which is more than the recommended / allowed range of years.

Leading and trailing blanks at every segment will be skipped as will be (allowed) leading zeroes. Hence "12-18-1" and " 2012 - 18 - 01 " and even " 2012 - 18 - 0x00001 " will get the same interpretation.

This function does only rudimentary checks to the ranges in the date string ((20)08 .. 2169, 1..12, 1..31). If one check fails or the string can't be interpreted 0 is returned. Fields in \ dat may have been changed nevertheless. The field `wd` (week day) in `dat` is set to 0 (illegal / unknown) instead of the correct (1..7) value in case of failure.

Parameters

dat	pointer to the result structure; may be changed partially even if the parse was not successful
s	the RAM string containing the date
si	the first index of the date in s

Returns

the index of the first character after the parsed time in case of success or 0 else

2.21.2.5 uint8_t searchFirstToken (char *s[],* uint8_t *si,* uint8_t *len*)

Search the first token in a short (RAM) string.

This function returns the start index (>=si) of the token found in string `s`. The begin of a token may be any non control code except space, comma, semicolon and equals (,;=).

255 is returned if no token was found before [len-1].

This function does almost the same as searchTokenStart() except that it skips over 0-codes.

The rationale for this 0-code skipping is that UART input may get zeros from PC emulated terminals when this PC enters or leaves energy saving mode. Using searchTokenStart() after those situations would swallow the next command line entry.

The consequence is, of course, that parameter `len` must be correct. The too big `len` for 0-terminated string idiom fails with this function.

Parameters

s	the RAM string to search in
si	the index to start search (usually 0 for first token)
len	the maximum length of s to search in

2.21.2.6 uint8_t **searchTokenStart** (char *s[]*, uint8_t *si*, uint8_t *len*)

Search the next token in a short (RAM) string.

This function returns the start index (>=si) of the token found in string `s`. The begin of a token may be any non control code except space, comma, semicolon and equals (,;=).

255 is returned if no token was found before [len-1] or a 0-code. This 0-code condition is the only difference to searchFirstToken().

Parameters

s	the RAM string to search in
si	the index to start search
len	the maximum length of s to search in

See also

searchTokenEnd

2.21.2.7 uint8_t **searchTokenEnd** (char *s[]*, uint8_t *si*, uint8_t *len*)

Search the the end of a token in a short (RAM) string.

This function returns the index of the first character following the the token starting at [si] in string `s`. It returns the `len` if no end was found before. The return value is <= `len`.

Any control character, space, comma, semicolon and equals (,;=) is considered as end of a token.

This function's return value (if < len) is a good start for the next token search by

```
searchTokenStart(s, retval, len)
```

.

Parameters

s	the RAM string to search in
si	the (found) token's begin index
len	the maximum length of s to search in

See also

searchTokenStart

2.21.2.8 uint8_t **searchTokenIn** (char *s[]*, uint8_t *si*, uint8_t *se*, char const *const * *list*, char const *const * *list2*)

Match a (short) token in a (RAM) string to a list of flash strings.

This function returns the start index of the token's first match in a flash array `list` of flash strings. The array list must end with a NULL, like in example

```
 char* systemCommands[]) = {comHelp, comWDlong, NULL};
```

The other entries, comHelp and comWDlong in the example, are 0-terminated strings in flash memory (PROGME-M).

The same applies to the second list `list2`. No list shall be longer than INDEX_OFFSET_LIST2.

The return value is the first match's index (range 0 .. llen-1) or 255 if no match could be found or is possible.

The match is made to the respective beginnings of the flash strings. A positive match will occur also if the flash string is longer. To avoid or recognise a hit to an ambiguous abbreviation a second search beyond the first hit is then made. 254 is returned if a second ambiguous hit is found.

The comparison is case insensitive. The token must not contain any control or white space characters.

Parameters

s	the RAM string containing the token to search for
si	the token's start index in s
se	the token's end index + 1 in s
list	the flash array of flash strings to match to (ending with NULL)
list2	the second flash array of flash strings to match to (ending with NULL) 120 is added to the return match index to distinguish the result from a match in `list`

Returns

the index of a match in list; or the index + INDEX_OFFSET_LIST2 of a match in list 2; or 255 if no match; or 254 if the token in s was too short to match unambiguously

See also

searchTokenStart
searchTokenEnd

2.21.2.9 char∗ getFirstSVNtokenP (char ∗ *dest,* char const ∗ *src,* uint8_t *mxLen*)

Copy the first SVN token or a full string from program space.

This function copies the first SVN-token — i.e. Subversion keyword's (first) value — from `src` to `dest` plus a trailing 0. Returned is the address of the last character in `dest` modified (i.e. the 0). In the example

"$Dáte: 2012-12-19 14:13:57 +0100 (Mi, 19 Dez 2012) $"

the first SVN token would be "2012-12-19".

The operation will stop after `mxLen` characters transferred resp. modified. The `mxLen` count includes the 0 appended.

If `src` does not begin with a $ `src` will be transferred from start to first $ or white space (or end).

Parameters

dest	the destination to modify (in RAM)
src	the source (the SVN tag) to copy from (in flash memory)
mxLen	the maximum number of characters to modify in `dest` including the trailing 0 appended. The maximum advance of the return value to parameter `dest`

Returns

dest + number of characters modified; NULL only if dest is NULL

2.21.2.10 uint8_t **asDigit** (uint8_t *c*)

Recognise one digit.

This helper function returns the digit value 0..9 for characters '0'..'9' and 10..15 for characters 'a'..'f' or 'A'..'F'. If the parameter c is in neither range 16 is returned. Hence result & 16 means invalid.

Parameters

c	the character

Returns

0..10..15 if c is a valid decimal or hexadecimal digit and 16 otherwise

2.21.2.11 uint8_t **parse2hex** (uint8_t ∗ *res,* char *s[ ],* uint8_t *si*)

Parse a two digit hexadecimal number.

This function tries to interpret a part of the string `s` starting at `si` as one or two digit hexadecimal number. Returned is the index of the first character after that number in case of success and 0 otherwise.

Leading blanks will be skipped.

Parameters

res	pointer to the result; changed only if a hexadecimal number was parsed successfully
s	the RAM string containing the 2 digit hex (00..FF/ff)
si	the first digit's index in s

Returns

the index of the first character after the parsed number in case of success or 0 else

See also

parseByteNum()
asDigit()

Examples:

main.c.

2.21.2.12 uint8_t **parseByteNum** (uint8_t ∗ *res,* char *s[ ],* uint8_t *si*)

Parse a one byte number.

This function tries to interpret a part of the string `s` starting at `si` as an eight bit (one byte) number. Returned is the index of the first character after that (parsed) number in case of success and 0 otherwise.

Leading blanks will be skipped. One sign character '+' or '-' is accepted. A minus will complement the result, notwithstanding it being declared unsigned or eventually parsed as hex. Leading zeroes don't, of course change, the result and they do not lead to octal interpretation.

A leading 0x or 0X will have the following one or two characters be interpreted as hexadecimal digits not counting leading zeros after the 'x'. The parsing then will stop at the latest after the second digit even if the character following (where the returned value points to) is a legal hex digit character.

Parameters

res	pointer to the result; changed only if a one byte number (0..255 or 0x0 .. 0xFF) was parsed successfully
s	the RAM string containing the digit(s)
si	the first digit's index in s

Returns

the index of the first character after the parsed number in case of success or 0 else

See also

parseWordNum()
parseDWordNum()
parse2hex()

2.21.2.13 uint8_t parseWordNum (uint16_t ∗ *res,* char *s[ ],* uint8_t *si*)

Parse a two byte number.

This function tries to interpret a part of the string `s` starting at `si` as a 16 bit (type uint16_t) number. Returned is the index of the first character after that (parsed) number in case of success and 0 otherwise.

Leading blanks will be skipped. One sign character '+' or '-' is accepted. A minus will complement the result, notwithstanding it being declared unsigned or eventually parsed as hex. Leading zeroes don't, of course change, the result and they do not lead to octal interpretation.

A leading 0x or 0X will have the following one to four characters be interpreted as hexadecimal digits not counting leading zeros after the 'x'. The parsing then will stop at the latest after the forth digit even if the following character (where the returned value points to) is a legal hex digit character.

Parameters

res	pointer to the result; changed only if a number was parsed successfully
s	the RAM string containing the digit(s)
si	the first digit's index in s

Returns

the index of the first character after the parsed number in case of success or 0 else

See also

parseByteNum
parseDWordNum

Examples:

main.c.

2.21.2.14 uint8_t parseDWordNum (uint32_t ∗ *res,* char *s[ ],* uint8_t *si*)

Parse a four byte number.

This function tries to interpret a part of the string `s` starting at `si` as a 32 bit (type uint32_t) number. Returned is the index of the first character after that (parsed) number in case of success and 0 otherwise.

Leading blanks will be skipped. One sign character '+' or '-' is accepted. A minus will complement the result, notwithstanding it being declared unsigned or eventually parsed as hex. Leading zeroes don't, of course change, the result and they do not lead to octal interpretation.

A leading 0x or 0X will have the following one to eight characters be interpreted as hexadecimal digits not counting leading zeros after the 'x'. The parsing then will stop at the latest after the eighth digit even if the character following (where the returned value points to) is a legal hex digit character.

Parameters

res	pointer to the result; changed only if a number was parsed successfully
s	the RAM string containing the digit(s)
si	the first digit's index in s

Returns

the index of the first character after the parsed number in case of success or 0 else

See also

parseByteNum
parseWordNum

2.21.2.15 uint8_t **isContainedIn** (char *s[ ]*, const uint8_t *c*)

Check if a character is contained in a RAM string.

This function return true if the character `c` is not 0 and contained in the 0-terminated string `s`.

Parameters

s	the (0-terminated) RAM string to be searched for `c`
c	the (non 0) character looked for

Returns

0 : `c` was not found in `s`, `c` : `c` was found in `s`

2.21.2.16 uint8_t **isContainedIn_P** (char *s[ ]*, const uint8_t *c*)

Check if a character is contained in a flash string.

This function return true if the character `c` is not 0 and contained in the 0-terminated string `s`. `s` must be in flash memory.

Parameters

s	the (0-terminated) string in flash / program memory to be searched for `c`
c	the (non 0) character looked for

Returns

0 : `c` was not found in `s`, `c` : `c` was found in `s`

2.22 Text blocks and utilities

2.22.1 Overview

Predefined text blocks and handling utilities are provided for system and application software. Whenever possible constant strings ar held in program (flash) memory only.

See also the common types and helpers

Defines

- #define comAutomationNum

    command: [-stop | -start] PLC cycles (flash text)
- #define comBootNum

    command: boot -load | -reset (flash text)
- #define comDateNum

    command: show / set (local) date
- #define comDHCPNum

    command: show / restart DHCP
- #define comDirlistNum

    command: dirList [-option] [dirPath] memory card directory
- #define comDNSNum

    command: resolve a name by DNS
- #define comENCNum

    command: ENC28J60 network interface (same as NICont)
- #define comEthAddrNum

    command: show address (ARP) info (flash text)
- #define comHelpNum

    command: show help (flash text)
- #define comIPconfigNum

    command: show IP configuration info (flash text)
- #define comIPdefAdNum

    command: show / set IP address (flash text)
- #define comMacAddrNum

    command: show / set MAC address (flash text)
- #define comModbusNum

    command: modbus [-stop -off -reset | ...
- #define comNICNum

    command: NICont [-off | -on | -reset | -restart]
- #define comNTPNum

    command: show / set NTP state and configuration
- #define comPLContrlNum

    command: [-stop | -start] PLC cycles (flash text)
- #define comRunInfoNum

    command: show run time info (flash text)
- #define comSMCNum

    command: SMCard [-on | -start | reset | -rest...
- #define comTelnetNum

    command: telnet [-stop]
- #define comTimeNum

    command: show / set (local) time

- #define comTypeFileNum

  command: typeFile filename display a file
- #define comUARTNum

  command: show / set UART flow control (flash text)
- #define comVersionNum

  command: show version info (flash text)
- #define comWatchDNum

  command: watchdog [abort | sharp | lenient | normal]
- #define comWDogNum

  command: wdog 60 ms | 120 ms | 250 ms
- #define comZoneNum

  command: show / set time zone and DST state
- #define optAbortNum

  options: abort a hard reset
- #define optAlternNum

  options: use or restart with alternative
- #define optAmbigousNum

  options: option given ambiguously
- #define optApplicNum

  options: application related
- #define optDebugNum

  options: debug lengthy output for trouble shooting
- #define optFastNum

  options: fast fast(er) speed
- #define optFlowCnNum

  options: ask or set flow control behaviour
- #define optHelpNum

  options: help command specific help
- #define optInfoNum

  options: inform normal / informing output
- #define optLenientNum

  options: lenient more forgiving
- #define optLoadNum

  options: perform load operation or set load mode
- #define optLogHereNum

  options: use this device as log output
- #define optNoLogsNum

  options: do not use this device as log output
- #define optNormalNum

  options: normal standard behaviour / speed
- #define optNotGivenNum

  options, commands, parameters: no option given / set
- #define optOffNum

  options: off turn off / stop
- #define optOnNum

  options: on turn on / start
- #define optOptionNum

  options: options related
- #define optPrimarNum

  options: use or restart with primary resource
- #define optQuestNum

options: ? like -help

- #define optQuietNum

  options: quiet like -silent
- #define optReadNum

  options: perform read operation or set read mode
- #define optResetNum

  options: reset like -stop or -restart (it depends)
- #define optRestartNum

  options: restart like -start (in most cases)
- #define optSharpNum

  options: sharp more exact / less forgiving
- #define optSilentNum

  options: silent no or only urgent output
- #define optSlowNum

  options: slow slow(er) speed
- #define optStartNum

  options: start like -on (in most cases)
- #define optStopNum

  options: stop like -off (in most cases)
- #define optSystemNum

  options: system related
- #define optVerboseNum

  options: verbose lengthy output
- #define optWriteNum

  options: perform write operation or set write mode

Functions

- char ∗ getSomeCharsP (char ∗dst, char const ∗src, uint8_t mxLen)

  Copy some characters from program space to a string.

Variables

- char const arpEmpty []

  " no valid entries in ARP table \n\n" flash text building block
- char const const arpEntries []

  " ARP entries " flash text building block
- char const comAutomation []

  command: [-stop | -start] PLC cycles (flash text)
- char const comBoot []

  command: boot -load | -reset (flash text)
- char const comDate []

  command: show / set (local) date
- char const comDHCP []

  command: show / restart DHCP
- char const comDirlist []

  command: dirList [-option] [dirPath] memory card directory
- char const comDNS []

  command: resolve a name by DNS
- char const comENC []

command: ENC28J60 network interface (same as NICont)
- char const comEthAddr [ ]

  command: show address (ARP) info (flash text)
- char const comHelp [ ]

  command: show help (flash text)
- char const comIPconfig [ ]

  command: show IP configuration info (flash text)
- char const comIPdefAd [ ]

  command: show / set IP address (flash text)
- char const comMacAddr [ ]

  command: show / set MAC address (flash text)
- char const commAmbig [ ]

  command report: ambiguous (flash text)
- char const comModbus [ ]

  command: modbus [-stop -off -reset | ...
- char const commWrong [ ]

  command report: wrong (flash text)
- char const comNIC [ ]

  command: NICont [-off | -on | -reset | -restart]
- char const comNTP [ ]

  command: show / set NTP state and configuration
- char const comPLContrl [ ]

  command: [-stop | -start] PLC cycles (flash text)
- char const comRunInfo [ ]

  command: show run time info (flash text)
- char const comSMC [ ]

  command: SMCard [-on | -start | reset | -rest...
- char const comTelnet [ ]

  command: telnet [-stop] : Telnet [close]
- char const comTime [ ]

  command: show / set (local) time
- char const comTypeFile [ ]

  command: typeFile filename display a file
- char const comUART [ ]

  command: show / set UART options
- char const comVersion [ ]

  command: show version info (flash text)
- char const comWatchD [ ]

  command: watchdog [abort | sharp | lenient | normal]
- char const comWDog [ ]

  command: wdog 60 ms | 120 ms | 250 ms
- char const comZone [ ]

  command: show / set time zone and DST state
- char const encEthSta [ ]

  " ENC/Eth st. : " flash text building block
- char const helpHeader [ ]

  Headline for system commands overview (help list)
- char const helpUserCm [ ]

  Headline for application commands overview (help list)
- char const ip4Add_is [ ]

  " IP4 address : " flash text building block

- char const ipConDefR [ ]

  " def. router : " flash text building block
- char const ipConDefS [ ]

  " def. set " flash text building block
- char const ipConDHCs [ ]

  " DHCP set " flash text building block
- char const ipConDHCu [ ]

  " use DHCP " flash text building block
- char const ipConDNSa [ ]

  " DNS address : " flash text building block
- char const ipConIpAd [ ]

  " IP4 address : " flash text building block
- char const ipConNMsk [ ]

  " IP4 netmask : " flash text building block
- char const ipConNTPa [ ]

  " NTP address : " flash text building block
- char const ipDefault [ ]

  " IP4 default : " flash text building block
- char const const macAdd_is [ ]

  " MAC address : " flash text building block
- char const noUserCLI [ ]

  report: no user CLI (flash text)
- char const optAbort [ ]

  options: abort a hard reset
- char const optAltern [ ]

  options: use or restart with alternative
- char const optAmbig [ ]

  option report: ambiguous (flash text)
- char const optApplic [ ]

  options: application related
- char const optDebug [ ]

  options: debug lengthy output for trouble shooting
- char const optFast [ ]

  options: fast fast(er) speed
  ";
- char const optFlowCn [ ]

  options: ask or set flow control behaviour
- char const optHelp [ ]

  options: help command specific help
- char const optInfo [ ]

  options: inform normal / informing output
- char const optionHeader [ ]

  *options: * * * Command options*
- char const optLenient [ ]

  options: lenient more forgiving
- char const optLoad [ ]

  options: perform load operation or set load mode
- char const optLogHere [ ]

  options: use this device as log output
- char const optNoLogs [ ]

  options: do not use this device as log output

- char const optNormal [ ]

  options: normal standard behaviour / speed
- char const optOff [ ]

  options: off turn off / stop
- char const optOn [ ]

  options: on turn on / start
- char const optOption [ ]

  options: options related
- char const optPrimar [ ]

  options: use or restart with primary resource
- char const optQuest [ ]

  options: ? like -help
- char const optQuiet [ ]

  options: quiet like -silent
- char const optRead [ ]

  options: perform read operation or set read mode
- char const optReset [ ]

  options: reset like -stop or -restart (it depends)
- char const optRestart [ ]

  options: restart like -start (in most cases)
- char const optSharp [ ]

  options: sharp more exact / less forgiving
- char const optSilent [ ]

  options: silent no or only urgent output
- char const optSlow [ ]

  options: slow slow(er) speed
  ";
- char const optStart [ ]

  options: start like -on (in most cases)
- char const optStop [ ]

  options: stop like -off (in most cases)
- char const optSystem [ ]

  options: system related
- char const optVerbose [ ]

  options: verbose lengthy output
- char const optWrite [ ]

  options: perform write operation or set write mode
- char const optWrong [ ]

  option report: wrong (flash text)
- char const sepLoB [ ]

  Blanks and left opening brace.
- char const sepRcB [ ]

  Right closing brace and blanks.
- char const sysRunSectorN [ ]

  report: LF SMC sector buffered: 0x
- char const systAut [ ]

  The author of weAutSys.
- char const systBld [ ]

  The build date and time.
- char const systBye [ ]

  An farewell (abort) line with three leading feeds and the system name.

- char const systCop [ ]

  weAutSys's copyright notice.
- char const systDat [ ]

  The system's last modification date.
- char const ∗const systemCommands [ ]

  flash array of the (flash) system command definitions
- char const ∗const systemOptions [ ]

  List and definitions of command options.
- char const systMod [ ]

  The system's last modifier.
- char const systNam [ ]

  The name of the runtime system weAutSys.
- char const systRev [ ]

  The system's revision.
- char const systWlc [ ]

  A greeting line with three leading feeds and the system name.
- char const wdSetLeni [ ]

  report: watchdog set long / lenient (flash text)r
- char const wdSetNormal [ ]

  report: watchdog set normal (flash text)r
- char const wdSetSharp [ ]

  report: watchdog set sharp (flash text)
- char const wdSetShort [ ]

  report: watchdog set short (flash text)

2.22.2 Function Documentation

2.22.2.1 char∗ getSomeCharsP (char ∗ *dst,* char const ∗ *src,* uint8_t *mxLen*)

Copy some characters from program space to a string.

This function copies up to `mxLen-1` characters from `src` to `dst` and appends a trailing 0. Returned is the address of the last character in `dst` modified (the 0's address).

The operation will stop after `mxLen` characters transferred resp. modified or if a terminating 0 if found in `src`.

Parameters

dst	the destination to modify (in RAM, not null!)
src	the source (string) to copy from (in flash memory, not null!)
mxLen	the maximum number of characters to modify in `dst` including the trailing 0 appended. The maximum advance of the return value to parameter `dst`

Returns

dst + number of characters modified; NULL only if `dst` is NULL

2.22.3 Variable Documentation

2.22.3.1 char const systNam[ ]

The name of the runtime system weAutSys.

This is a string in flash memory.

See also

systWlc

2.22.3.2 char const systWlc[ ]

A greeting line with three leading feeds and the system name.

This is a string in flash memory.

See also

systNam
systBye

2.22.3.3 char const systBye[ ]

An farewell (abort) line with three leading feeds and the system name.

This is a string in flash memory.

See also

systNam
systWlc

2.22.3.4 char const systRev[ ]

The system's revision.

It is the SVN revision of the (this) file system.h. If not treated with beautifying tools like de.frame4j.CVSkeys (.java) prior to make it will be the full SVN tag in Dollars (

tagname:

token

).

This is a string in flash memory.

See also

getFirstSVNtokenP(char∗, prog_char∗, uint8_t)

2.22.3.5 char const systDat[ ]

The system's last modification date.

It is in fact the SVN date of the (this) file system.h. If not treated with beautifying tools like de.frame4j.CVSkeys (.java) prior to the make process it will be the full SVN tag in Dollars (

tagname:

token token token

).

Un-beautified — the usual vase — this is a horrible date time zone (text) format; the first token is the date like 2011-03-11 (yyyy-mm-dd). This can be extracted by getFirstSVNtokenP()

This is a string in flash memory.

2.22.3.6 char const systMod[ ]

The system's last modifier.

It is the SVN user name of the person who did the latest modification of the (this) file system.h. If not treated with beautifying tools like

```
de.frame4j.CVSkeys
```

(.java, see frame4j.de) prior to make it will be the full SVN tag in Dollars (

tagname:

token

).

de.frame4j.CVSkeys can be configured to replace SVN or system account names with the human's name, like e.g.

- albrecht by Albrecht Weinert
- ralf by Ralf Seidel etc.

This is a string in flash memory.

See also

getFirstSVNtokenP(char∗, prog_char∗, uint8_t)

2.22.3.7 char const systAut[ ]

The author of weAutSys.

It is the author's name and his personal domain.

This is a string in flash memory.

See also

getSomeCharsP(char∗, prog_char∗, uint8_t)

2.22.3.8 char const systCop[ ]

weAutSys's copyright notice.

This is a string in flash memory.

See also

getSomeCharsP(char∗, prog_char∗, uint8_t)

2.22.3.9 char const systBld[ ]

The build date and time.

It is the word "build" followed by the date and time where the C-preprocessor rolled over the (this) file system.h. This will be time where the weAutSys runtime was build (if you get the text from a living system.

This is a string in flash memory.

See also

getSomeCharsP(char∗, prog_char∗, uint8_t)

2.22.3.10 char const sepLoB[ ]

Blanks and left opening brace.

It is " (".

This is just a building block for Strings held in flash memory.

See also

getSomeCharsP(char∗, prog_char∗, uint8_t)

2.22.3.11 char const sepRcB[ ]

Right closing brace and blanks.

It is ") ".

This is just a building block for Strings held in flash memory.

See also

getSomeCharsP(char∗, prog_char∗, uint8_t)

2.22.3.12 char const∗ const systemOptions[ ]

List and definitions of command options.

This is the flash array of the (flash) option definitions. Options are recognised by setCliLine() if prefixed by - (minus) and given as first parameter of a command, like e.g. "help -options". Like with commands, options may be abbreviated if the abbreviation is not ambiguous, as would be -res instead of -reset respectively -restart.

As commands options are not case sensitive.

Unrecognised options may be (silently) or may be not ignored by system and application commands.

See also

setCliLine()

2.22.3.13 char const helpUserCm[ ]

Headline for application commands overview (help list)

This flash text must be the first entry in userCommands[] if user software chooses to register application specific commands.

Examples:

main.c.

2.22.3.14 char const helpHeader[ ]

Headline for system commands overview (help list)

This flash text is the first entry in systemCommands[].

2.23 Persistent storage

2.23.1 Overview

Utility functions and threads are provided for persistent storage. The µ-controller ATmega1284P has 4K byte EEPROM (100T erase cycles life). The board weAut_01 has a slot for small memory cards.

Both can be used for persistent storage. To hold configuration data to survive power off the EEPROM is clearly the better choice.

Files

- file persist.h

  weAutSys (weAut_01) utility / library functions for persistent storage

Data Structures

- struct eeOp_data_t

  The data structure for an EEPROM bulk operation thread.

Defines

- #define DEFAULT_EEPROM_CONF_ADD 64

  Default address of EEPROM configuration data.
- #define EE_CONF_ADD(elem)

  The EEPROM address of a configuration element.
- #define EEPROM_POINTER2_EE_CONF (EEP_SIZE - 64)

  EEPROM address of (address of) EEPROM configuration data.

Functions

- ptfnct_t eeOperationThread (eeOp_data_t ∗eeOpData)

  The EEPROM (bulk) write system thread; the thread function.
- uint8_t eeReadByte (uint8_t ∗readValue, uint16_t eeAddr)

  Read one EEPROM cell to RAM.
- uint8_t eeReadBytes (uint16_t eeAddr, uint8_t ∗dest, uint8_t bufferLength)

  Read EEPROM cells.
- uint8_t eeWriteByte (uint16_t eeAddr, uint8_t newValue)

  Write one EEPROM cell.
- uint8_t initEEthreadWrite (eeOp_data_t ∗eeOpData, uint16_t eeAddr, uint8_t ∗writeBuffer, uint8_t bufferLength)

  Initialise an EEPROM operation thread.

Variables

- uint16_t eeConfigAdd

  Address of EEPROM configuration data.
- eeOp_data_t eeOpData

  The data for an EEPROM bulk operation thread.

2.23.2 Define Documentation

2.23.2.1 #define EEPROM_POINTER2_EE_CONF (EEP_SIZE - 64)

EEPROM address of (address of) EEPROM configuration data.

This is the address of two consecutive bytes in EEPROM that hold the address of this devices EEPROM configuration data.

The rationale of this indirection is that that the entry pointed to by this address will have to be changed only if the writing of configuration data fails due to EEPROM wear out (after $>$ 100000 modifies).

See also

DEFAULT_EEPROM_CONF_ADD

2.23.2.2 #define DEFAULT_EEPROM_CONF_ADD 64

Default address of EEPROM configuration data.

This is the (fresh board default) value for the address of this devices EEPROM configuration data.

The value might be changed in future software releases to use other EEPROM locations on (then) older boards.

Hint: This value has to be used in (C-) sources and tool (avr-objcopy) settings respectively parameters to generate an EEPROM .hex file with individualized board configurations.

Hint 2: There is a (until April 2013 at least) avrdude bug that erases all EEPROM cells below the first address of a small EEPROM .hex file. As long as this problem persists DEFAULT_EEPROM_CONF_ADD should be quite low and must be lower than EEPROM_POINTER2_EE_CONF.

See also

EEPROM_POINTER2_EE_CONF

2.23.2.3 #define EE_CONF_ADD(*elem*)

The EEPROM address of a configuration element.

This macro calculates the EEPROM address of an element in a conf_data_t structure. This is done without without any precondition checks. The most important condition is eeConfigAdd being not NULL.

Parameters

elem	the name of a conf_data_t element

Returns

its EEPROM address in an eeConfigAdd based conf_data_t structure

2.23.3 Function Documentation

2.23.3.1 uint8_t eeReadByte (uint8_t ∗ *readValue,* uint16_t *eeAddr*)

Read one EEPROM cell to RAM.

The operation is a bit slow (compared to RAM access) but the function returns immediately.

The function fails if another EEPROM (or Flash write) operation is running. It must, of course, never ever be used spin waiting for success.

Parameters

eeAddr	the EEPROM address to read (0 .. 4095)
readValue	pointer to the (RAM) byte to put the value read in

Returns

0: fail; 1: success

See also

eeWriteByte
eeReadBytes

2.23.3.2 uint8_t **eeReadBytes** (uint16_t *eeAddr,* uint8_t ∗ *dest,* uint8_t *bufferLength*)

Read EEPROM cells.

This function reads a number of consecutive EEPROM cells. Reading EEPROM is four to five times slower than RAM access.

The function fails and returns immediately if another EEPROM (or Flash write) operation is running. It must, of course, never ever be used spin waiting for success.

Parameters

eeAddr	the EEPROM address to read (0 .. 4095)
dest	the RAM address to put the EEPROM data to
bufferLength	the number of bytes to handle 1..255

Returns

the number of bytes transfered (may be shorter than bufferLenght or 0 in case of failure or inadequate parameters

See also

eeReadByte

2.23.3.3 uint8_t **eeWriteByte** (uint16_t *eeAddr,* uint8_t *newValue*)

Write one EEPROM cell.

This function writes the EEPROM cell. If the (slow and wearing) write is necessary it starts the write procedure for the new changed content. The operation is quite slow (compared to RAM access) but the function returns immediately either after failure or after just starting the (3 ms) EEPROM write procedure.

The function fails if another EEPROM (or Flash write) operation is running. It should not be used by application software and it must, of course, never ever be called spin waiting for success.

Parameters

eeAddr	the EEPROM address to (read and) write
newValue	the (new) value to write in the EEPROM cell[eeAddr]

Returns

0: fail; 1: success

See also

eeReadByte

2.23.3.4 uint8_t initEEthreadWrite (eeOp_data_t * *eeOpData*, uint16_t *eeAddr*, uint8_t * *writeBuffer*, uint8_t *bufferLength*)

Initialise an EEPROM operation thread.

Before scheduling the EEPROM operation thread for first start of a bulk operation it's thread data (eeOpData) must be initialised.

Calling this or another initialising function while a thread using the structure (eeOpData) is operating will have unwanted if not catastrophic results.

Parameters

eeOpData	the thread's data
eeAddr	the EEPROM address to (read and) write
writeBuffer	the source of the new EEPROM data
bufferLength	the number of bytes to handle

Returns

the number of bytes to be handled by the thread (may be shorter than bufferL)

See also

initEEthreadRead

2.23.3.5 ptfnct_t eeOperationThread (eeOp_data_t * *eeOpData*)

The EEPROM (bulk) write system thread; the thread function.

According to the task set in eeOpData this thread will transfer data from RAM to EEPROM.

At first schedule(s) this thread blocks as long as it has not gained access to the EEPROM.

After the operation has begun it will yield after every step requiring an EEPROM write and after every eight steps without the need to modify the EEPROM content.

As one EEPROM write requires 3.4 ms, it could be considered to schedule every forth time in a 1 ms cycle or at lower frequency.

This thread is best scheduled by

```
initEEthreadWrite(&eeOp_data, .....);
PT_WAIT_THREAD(callingParentThread, eeOperationThread(&eeOpData));
```

Parameters

eeOpData	pointer to the data of the EEPROM operation

Returns

PT_WAITING while waiting for EEPROM ready on first schedule(s); PT_YIELDED while waiting for EEPROM ready after a modify step; PT_YIELDED after every 8th read only step; PT_ENDED when ready

2.23.4 Variable Documentation

2.23.4.1 eeOp_data_t eeOpData

The data for an EEPROM bulk operation thread.

It is quite possible to have more than one active protothread (function eeOperationThread) at the same time. In that case each must use its own eeOp_data_t structure.

On the other hand it seems clever to restrict EEPROM access to just one thread at one time. In that case they all can use the same eeOp_data_t structure, optionally using noOfWrite as a signal for availability. This eeOpData are used by weAutSys system software at initialisation time only. Application software is free to use it afterwards (i.e. in any thread except appInitThreadF).

2.23.4.2 uint16_t eeConfigAdd

Address of EEPROM configuration data.

This is the address of this device's EEPROM configuration data.

This EEPROM address may be in the range 64 .. (EEPROMsize - 64).

It is read at start-up from EEPROM[EEPROM_POINTER2_EE_CONF]. If outside said rage a default configuration will be set up at EEPROM address 64.

See also

conf_data_t
EE_CONF_ADD

2.24 + + Process I/O and HMI functions + +

2.24.1 Overview

Process I/O and human interface functions are, of course, the field of the automation / application programmer. This task is facilitated by weAutSys providing the basic hardware abstraction functions to access the I/O periphery.

Process I/O here means the device's (weAut_01's) protected I/O to the "field" through clamps and connectors and not the internal (un-protected) extension and instrumentation ports.

HMI here means the device's (weAut_01's) visible display LEDS and its (2) switches and not any HMI through serial and Ethernet communication links.

Modules

- Digital output
- Digital / Analogue input
- HMI elements

Files

- file proc_io.h

  weAutSys'/weAut_01's (low level) system calls and services for process I/O and HMI

2.25 Digital output

2.25.1 Overview

weAut_01 (weAutSys' target hardware) has 8 digital output (DO) channels (LV, 100mA, protected). Those outputs are supervised with respect to overload and over-temperature.

The allowed range for the load voltage (LV) is 7..28V, the nominal values being either 12V or 24V. The LV is supervised, too.

Functions

- uint8_t actDOdriver (void) __attribute__((always_inline))

  DO digital (process) output: actual value.
- uint8_t doDriverEnabled (void) __attribute__((always_inline))

  Digital (process) output DO driver enabled.
- uint8_t doDriverOK (void) __attribute__((always_inline))

  Digital (process) output DO driver OK.
- void enableDOdriver (uint8_t state) __attribute__((always_inline))

  Enable the digital (process) output DO driver.
- uint8_t prvDOdriver (void) __attribute__((always_inline))

  DO digital (process) output: previous value.
- void toDOdriver (uint8_t value)

  Output to digital (process) output DO.

Variables

- uint8_t lowLV

  Load voltage low.

2.25.2 Function Documentation

2.25.2.1 void **toDOdriver** (uint8_t *value*)

Output to digital (process) output DO.

This function outputs an eight bit value to the standard digital process output.

On weAut_01 (weAut_00) this is the complete SPI 1 output function to the protected DO driver and its 8 green (ordered output) status LEDs.

The `value` goes direct to the 8 green DO LEDs. The state driven to the DO outputs further depends on the state of the driver (enable / disable, fault) and on the level of its supply -- the load voltage (LV) or "Lastspannung" in German.

On other platforms (arduMega2560 and the like) the value will be output to the eight bit port configured. If no DO is configured (marcro DO_0_PORT undefined) no output will be done (but the states are hold).

Parameters

value	8 bits for 8 green LEDs, upper row, located in the same column as the corresponding DO clamps

See also

enableDOdriver
lowLV

toDiLEDs

Examples:

main.c.

2.25.2.2 uint8_t **actDOdriver** (void)

DO digital (process) output: actual value.

This function returns the actual value of the DI LEDs.

See also

toDOdriver

2.25.2.3 uint8_t **prvDOdriver** (void)

DO digital (process) output: previous value.

This function returns the actual value of the DI LEDs.

See also

toDOdriver

2.25.2.4 void **enableDOdriver** (uint8_t *state*)

Enable the digital (process) output DO driver.

This function allows all 8 DO output channels to be turned off respectively on.

On weAut_01 (weAut_00) this will control the protected DO driver (9..28V "Lastspannung") and does in no way influence the output value set by `toDOdriver(uint8_t)`, which will be still displayed by the (8 green) DO LEDs.

The off state will be signaled by the (red) DO disable LED.

Also, turning off is the procedure to reset any output driver's fault state, be it on over temperature or over current. Turning off on a fault state will probably not make the situation worse. A fault condition turns the output driver off anyway.

On other platforms this will be used to mask out the the output value set by `toDOdriver(uint8_t)` respectively re-enable it.

See also

toDOdriver

Parameters

state	true: enable; false: disable

Examples:

main.c.

2.25.2.5 uint8_t doDriverEnabled (void)

Digital (process) output DO driver enabled.

See also

toDOdriver return state true: enabled; false: disabled

Examples:

main.c.

2.25.2.6 uint8_t doDriverOK (void)

Digital (process) output DO driver OK.

On weAut_01 (weAut_00) this function returns true if the protected DO driver (9..28V "Lastspannung") is neither in over current nor over temperature shut down state for at least one output channel.

On other platforms with a Port dedicated to DO this function returns true if the re-read out pin values equal the actual output. If that is not so there may be shorts to Vdd or Gnd or configuration problems.

In all other cases this function blindly returns true.

See also

toDOdriver return state true: OK; false: possible problems as described

Examples:

main.c.

2.25.3 Variable Documentation

2.25.3.1 uint8_t lowLV

Load voltage low.

Non 0 means a low load voltage supply (LV, German Lastspannung). In that case the digital output's (DO's) supply is below about 10.3V. This is also approximately the level where weAut_01's lower blue LEDs shine with only medium brightness. Lower side of the board means the RJ45 jack side.

At 10 V load voltage the processor's an all other periphery's supply is still guaranteed, even if the redundant supply should be absent. But the operation of 12V periphery is probably at risk and that of 24V periphery normally gone already.

Hint 1: If a more accurate control of LV is needed, its possible to tie an (DI) input as AI to LV (using it in the analogue mode then).

Hint 2: weAut_01 can have this LV threshold risen to (industrial automation) 24V mode by a simple wire bridge. See the hardware manual of your module.

Hint 3: `lowLV` is implemented as a counter counting up to 128 respectively down to 0 every millisecond according to LV $>$ about 10,3V. The 0-1-transition will set / un-set the red (test) LED, so red = on will indicate "LV low". This may slightly interfere with other usages of this red test LED during development / debugging phases.

After an intermediate use of the red (test) LED it can be restored to the proper (LV) state by `setStatusLed-Rd(lowLV).`

Hint 4: This variable should never be set by user / application software.

This feature is available at weAut_01 (weAut_00) only. Boards without "real" process signals and load supply handling can't have it.

See also

setStatusLedRd

2.26 Digital / Analogue input

2.26.1 Overview

weAut_01 (weAutSys' target hardware) has 8 protected input channels for +/-70V input voltage (surviving 250 Veff as absolute max. rating).

Every channel can be configured as

- digital input
  - in three threshold / hysteresis modes for different (12V / 24V) sensors
- analogue input
  - in three different single ended voltage ranges

Functions

- uint8_t actDI (void) __attribute__((always_inline))

  Actual digital (process) input DI.
- uint8_t dctDI (void) __attribute__((always_inline))

  Digital (process) input DI (direct)
- uint8_t filDI (void) __attribute__((always_inline))

  The final or filtered digital (process) input DI.
- uint8_t lbpDI (void) __attribute__((always_inline))

  Last before previous digital (process) input DI.
- void procDIcyc (void)

  Digital (process) input DI (system implementation)
- uint8_t prvDI (void) __attribute__((always_inline))

  Previous digital (process) input DI.
- void setAIchannels (uint8_t mask)

  Set the usage of channel(s) as AI instead of DI.

Variables

- uint8_t aiChannels

  Use channel(s) as AI instead of DI.
- uint8_t aiConvd

  Analogue input available.
- uint8_t aiResult [8]

  Analogue input results.
- uint8_t upDIthresh4hyst

  Shift DI thresholds up mask conditionally / larger hysteresis.
- uint8_t upDIthreshForce

  Shift DI thresholds up mask permanently.

2.26.2 Function Documentation

2.26.2.1 uint8_t dctDI (void)

Digital (process) input DI (direct)

This function returns the (8) digital inputs in one byte. The value by this function will be the immediate un-filtered read. Hence it is a low level function rarely (not to say never) to be used by user /application software directly.

For configurations without such inputs 0 will be returned. Channels used (in weAut_01 e.g.) for analogue input and disabled for digital reading (by DIDR0 the digital input disable register) will also return 0 in the respective bit position.

2.26.2.2 void procDIcyc (void)

Digital (process) input DI (system implementation)

Must not to be used by user / application software.

2.26.2.3 uint8_t actDI (void)

Actual digital (process) input DI.

This function returns the (8) digital inputs in one byte. The value by this function will be the actual un-filtered read, done by system software within the last one millisecond. Hence, the value is 0..1 ms old.

Hint: This statement is true for the normal configuration where all DI processing is done in the 1 ms system thread. If done more seldom by system (re-) configuration, apply the appropriate factor to all filter and timing values given for this function and the other related DI functions.

Hint 2: `actDI()`, `prvDI()`, `lbpDI()` and `dctDI()` are low level function rarely to be used by user / application software. Therefore `filDI()` is recommended.

See also

prvDI
lbpDI
filDI

2.26.2.4 uint8_t prvDI (void)

Previous digital (process) input DI.

This function returns the (8) digital inputs in one byte. The value here will be the previous un-filtered read, done by system software before the actual read. The value will be 1 ms older than `actDI()`, i.e. 1..2 ms old.

See also

actDI
lbpDI
filDI

2.26.2.5 uint8_t lbpDI (void)

Last before previous digital (process) input DI.

This function returns the (8) digital inputs in one byte. The value here will be the last before previous un-filtered read. The value will be 2 ms older than `actDI()`.

See also

actDI
prvDI
filDI

2.26.2.6 uint8_t filDI (void)

The final or filtered digital (process) input DI.

This function returns the (8) digital inputs in one byte. The value here will be the final value reflecting state values stable over the last three milliseconds by appropriate filter algorithm.

A stable input change will be reflected here after 2 .. 3 ms.

It is this (stable) value that is optionally used to enlarge the hysteresis (by slightly shifting the input thresholds),

See also

actDI
prvDI
dctDI
upDIthresh4hyst

Examples:

main.c.

2.26.2.7 void setAIchannels (uint8_t *mask*)

Set the usage of channel(s) as AI instead of DI.

For bits set 1 in mask the respective input channels are uses as analogue inputs AI (en lieu de DI).

This DI/AI configuration is only relevant with weAut_01 (weAut_00) where potential AI and DI share the same (rare) protected input clamps. Boards with direct access to (an abundance of) µController pins won't use that feature.

See also

aiChannels.

Examples:

main.c.

2.26.3 Variable Documentation

2.26.3.1 uint8_t upDIthreshForce

Shift DI thresholds up mask permanently.

For bits set 1 in this mask the respective DI channels permanently get higher input thresholds compared to those without such settings.

High settings: about 11V on; about 9V off.

Low settings: about 6V on; about 3V off.

This DI level setting works with weAut_01 (weAut_00) only. Boards without "real" digital process input have only CMOS logic inputs.

See also

upDIthresh4hyst
filDI

Examples:

main.c.

2.26.3.2 uint8_t upDIthresh4hyst

Shift DI thresholds up mask conditionally / larger hysteresis.

For bits set 1 in this mask byte the respective DI channels permanently get higher input thresholds on the condition of their `fiDI()` value being 1. This effectively widens this DI channels hysteresis.

Wide setting: about 11V on; about 3V off.

The hysteresis widening will only work if the respective bit in `upDIthreshForce` is **not** set.

This DI level / hysteresis settings work with weAut_01 (weAut_00) only. Boards without "real" digital process input have only CMOS logic inputs.

See also

upDIthreshForce
filDI

Examples:

main.c.

2.26.3.3 uint8_t aiChannels

Use channel(s) as AI instead of DI.

For bits set 1 in this mask byte the respective input channels are uses as analogue inputs AI (en lieu de DI).

This variable must not be set directly but changed by using `setAIchannels`.

Examples:

main.c.

2.26.3.4 uint8_t aiConvd

Analogue input available.

System software will set a bit if an analogue conversion, enabled by `aiChannels`, is complete. Converting all 8 channels (if told so by `aiChannels`) will take about 700µs.

2.26.3.5 uint8_t aiResult[8]

Analogue input results.

This array contains the latest AI conversion results (only for those channels enabled by `aiChannels`).

The results are restricted to 8 bit resolution which is quite appropriate regarding the use of weAut_01's use of its inputs designed as protected DIs in an analogue mode.

Examples:

main.c.

2.27 HMI elements

2.27.1 Overview

Human machine interface (HMI) here we mean display and control elements on the device visible and controllable for a person in front of it — and not the observing and controlling functions provided via serial and/or Ethernet communication links to locations more or less remote from the device.

weAut_01 (weAutSys' main target hardware) has

- two push button switches
  - one for (hard) reset and
  - one called "enter" for free use by user / application software
- eight green LEDs in proximity to the eight input channel clamps
  - either for the intended "state of input" use
  - or for free use by the software
- a red LED
  - either for the intended "load voltage down" use
  - or for free use by the software
- a green LED (free use by software)
- a green and a yellow LED in the Ethernet (RJ45) plug
  - either for the intended "link up / TX" use
  - or for free use by the software

All other (14 red, green, blue) LEDs are directly linked to digital output and supply voltage state without any extra influence of the software.

Defines

- #define enterKeyPressed(X)

  Enter key is (stable) pressed.
- #define enterKeyRel(X)

  Enter key is released.
- #define enterKeyReleased(X)

  Enter key is (stable) released.

Functions

- uint8_t actDiLEDs (void) __attribute__((always_inline))

  DI display LEDs: actual value.
- void offDiLEDs (void) __attribute__((always_inline))

  Turn DI display LEDs off.
- void onDiLEDs (void) __attribute__((always_inline))

  Turn DI display LEDs on.
- uint8_t prvDiLEDs (void) __attribute__((always_inline))

  DI display LEDs: previous value.
- void toDiLEDs (uint8_t value)

  Output to DI display LEDs.

Variables

- uint8_t pbFil

  Filtered Port B input.

2.27.2 Define Documentation

2.27.2.1 #define enterKeyRel(*X*)

Enter key is released.

This is the direct un-filtered state of the enter button (weAut_01).

0 (false) is returned if the button is pressed.

Non 0 is returned if the enter button is released.

On the weAut_01 board the button nearer to the Ethernet and serial connectors is (hard) reset). The one more far away is the "enter" button handled here.

Configurations without that key always return non 0 (true).

2.27.2.2 #define enterKeyReleased(*X*)

Enter key is (stable) released.

This is the filtered state of the enter key (weAut_01).

0 (false, OFF) is returned if the key is pressed.

Non 0 is returned if the enter key is released.

Configurations without that key always return 1 (non 0).

See also

enterKeyPressed

Examples:

main.c.

2.27.2.3 #define enterKeyPressed(*X*)

Enter key is (stable) pressed.

This is the filtered state of the enter key (weAut_01) .

Non 0 (true, ON) is returned if the key is released.

0 (false, OFF) is returned if the enter key is pressed.

Configurations without that key always return 0.

See also

enterKeyReleased

Examples:

main.c.

2.27.3 Function Documentation

2.27.3.1 void **toDiLEDs** (uint8_t *value*)

Output to DI display LEDs.

This function outputs an eight bit value to the process digital input (DI) status LEDs.

On weAut_01 (weAut_00) this is the complete SPI 1 output function to the DI-LED driver for its 8 green status LEDs (in neighborhood of the DI clamps).

Hint: Using the SPI interface is avoided if the result can be obtained by output enabling / disabling the LED driver. This (doubtful) optimisation allows (regular) blinking even if the SPI interface is blocked for other operations. This feature may be removed in later hardware versions.

On other platforms (arduMega2560 and the like) the value will be output to an eight bit port configured therefore. If no DI-LED port is configured (DI_LEDs0_PORT undefined) no output will be done (but the states are hold).

Parameters

value	8 bits for 8 green LEDs, upper row, located same column as the corresponding DI clamps

See also

toDOdriver

2.27.3.2 uint8_t **actDiLEDs** (void)

DI display LEDs: actual value.

This function returns the actual value of the DI LEDs.

See also

toDiLEDs()

2.27.3.3 uint8_t **prvDiLEDs** (void)

DI display LEDs: previous value.

This function returns the actual value of the DI LEDs.

See also

toDiLEDs()

2.27.3.4 void **offDiLEDs** (void)

Turn DI display LEDs off.

If the LEDs are off (==0) nothing is done.

If at least one is on this is a shorter equivalent for `toDiLEDs(0)`.

See also

toDiLEDs
onDiLEDs

2.27.3.5 void onDiLEDs (void)

Turn DI display LEDs on.

If not all LEDs are off (!=0) nothing is done.

If they are off this function is a (quick) equivalent for `toDiLEDs(` prvDiLEDs()).

See also

offDiLEDs

2.27.4 Variable Documentation

2.27.4.1 uint8_t pbFil

Filtered Port B input.

Port B0 == 1 means enter key on (weAut_01) is released.

See also

enterKeyReleased
enterKeyPressed

2.28 + + Low level system services + +

2.28.1 Overview

weAutSys provides a bundle of system function and variables mainly for internal use, that may well be used by application threads.

Compared to the system services the "low level" means that even more care must be taken on the usage conditions. And the implementation is in many cases directly dependent on target hardware and controller type.

Modules

- Software instrumentation
- Debugging aids
- System initialisation
- Basic I/O drivers (SPI)
- Basic serial communication drivers
- Basic Ethernet communications drivers

Files

- file ll_system.h

  weAutSys'/weAut_01' low level system calls and services

Defines

- #define abortBoot

  Abort command: go to bootloader (by human command)
- #define abortHMI

  Abort command: abort by human command entry.
- #define andAP(mask)

  Unset (boolean and operation) port bits.
- #define andBP(mask)

  Unset (boolean and operation) port bits.
- #define andCP(mask)

  Unset (boolean and operation) port bits.
- #define andDP(mask)

  Unset (boolean and operation) port bits.
- #define andOrBP(andMask, orMask)

  Unset and set port B bits.
- #define andOrCP(andMask, orMask)

  Unset and set port C bits.
- #define andPort(port, mask)

  Unset (boolean and operation) port-bits.
- #define dirPort(port)

  Direction port (by letter)
- #define fromPort(port)

  Re-read output from a port (by letter)
- #define inPort(port)

  Input port (by letter)
- #define inpPins(port, mask)

Set port pins as input.

- #define orAP(mask)

  Set (boolean or operation) port-bits.
- #define orBP(mask)

  Set (boolean or operation) port-bits.
- #define orCP(mask)

  Set (boolean or operation) port-bits.
- #define orDP(mask)

  Set (boolean or operation) port-bits.
- #define orPort(port, mask)

  Set (boolean or operation) port-bits.
- #define outPins(port, mask)

  Set port pins as output.
- #define PLCinRUN

  Status check: PLC is in Run.
- #define PLCinSTOP

  Status check: PLC is in Stop.
- #define PLCrun

  Abort command: run (i.e. no abort command at all)
- #define PLCstop

  Abort command: stop.
- #define setPort(port, value)

  Set all port-bits.
- #define TOKENPASTE(x, y)

  Concatenation helper macro x ## y.
- #define toPort(port, mask)

  Output to a port (by letter)
- #define WDtiOut

  Abort cause: (unexpected) watchdog timeout.
- #define xorPort(port, mask)

  Toggle (boolean xor operation) port-bits.

Functions

- void initPorts (void)

  low level reset type initialisation of ports
- void initProcIO (void)

  low level reset type initialisation of process I/O
- void initStatusLeds (uint8_t LEDgnStart, uint8_t LEDrdStart)

  Initialisation of status / test LEDs (if any)
- void initSystemRes (void)

  Initialise system resources after reset or restart.
- void initSystTiming (void)

  low level clock / timer / tick initialisation
- int main (void) __attribute__((OS_main))

  The system start.
- void setAbortCommand (uint8_t command) __attribute__((always_inline))

  Set the abort cause respectively command.

Variables

- uint8_t abortCommand

    The abort command respectively cause.
- uint8_t resetCauses

    The last reset cause(s)
- char const ∗ resetCauseText

    The last main cause as text.

2.28.2 Define Documentation

2.28.2.1 #define orAP(*mask*)

Set (boolean or operation) port-bits.

These low level helpers would / should hardly be called directly by user / application software.

2.28.2.2 #define andAP(*mask*)

Unset (boolean and operation) port bits.

These low level helpers would / should hardly be called directly by user / application software.

2.28.2.3 #define orBP(*mask*)

Set (boolean or operation) port-bits.

These low level helpers would / should hardly be called directly by user / application software.

2.28.2.4 #define andBP(*mask*)

Unset (boolean and operation) port bits.

These low level helpers would / should hardly be called directly by user / application software.

2.28.2.5 #define orCP(*mask*)

Set (boolean or operation) port-bits.

These low level helpers would / should hardly be called directly by user / application software.

2.28.2.6 #define andCP(*mask*)

Unset (boolean and operation) port bits.

These low level helpers would / should hardly be called directly by user / application software.

2.28.2.7 #define orDP(*mask*)

Set (boolean or operation) port-bits.

These low level helpers would / should hardly be called directly by user / application software.

2.28.2.8 #define andDP(*mask*)

Unset (boolean and operation) port bits.

These low level helpers would / should hardly be called directly by user / application software.

2.28.2.9 #define orPort(*port, mask*)

Set (boolean or operation) port-bits.

This low level helper would / should hardly be called directly by user / application software.

Parameters

port	The port's letter (A, B, C ...)
mask	The value to be ORed to PORTx

2.28.2.10 #define setPort(*port, value*)

Set all port-bits.

This low level helper would / should hardly be called directly by user / application software.

Parameters

port	The port's letter (A, B, C ...)
value	The value to be assigned to PORTx

2.28.2.11 #define andPort(*port, mask*)

Unset (boolean and operation) port-bits.

This low level helper would / should hardly be called directly by user / application software.

Parameters

port	The port's letter (A, B, C ...)
mask	The value to be ANDed to PORTx

2.28.2.12 #define xorPort(*port, mask*)

Toggle (boolean xor operation) port-bits.

This low level helper would / should hardly be called directly by user / application software.

Parameters

port	The port's letter (A, B, C ...)
mask	The value to be XORed to PORTx (by using the PINx trick)

See also

outPins

2.28.2.13 #define toPort(*port,* *mask*)

Output to a port (by letter)

This low level helper would / should hardly be called directly by user / application software.

Parameters

port	The port's letter (A, B, C ...)
mask	The value to be set resp. output to PORTx

See also

outPins

2.28.2.14 #define fromPort(*port*)

Re-read output from a port (by letter)

This low level helper would / should hardly be called directly by user / application software.

Parameters

port	The port's letter (A, B, C ...)

Returns

this expression is the re-read output from PORTx

2.28.2.15 #define inPort(*port*)

Input port (by letter)

This low level helper would / should hardly be called directly by user / application software.

Parameters

port	The port's letter (A, B, C ...)

Returns

this is I/O-Register PINx

See also

inpPins

2.28.2.16 #define inpPins(*port,* *mask*)

Set port pins as input.

This is a low level configuration function. It should hardly be called directly by user / application software (except with good knowledge and care).

Parameters

port	The port's letter (A, B, C ...)
mask	Bit values of 1 mark the bits reps. pins to be set to input

See also

outPins

2.28.2.17 #define outPins(*port*, *mask*)

Set port pins as output.

This is a low level configuration function. It should hardly be called directly by user / application software (except with good knowledge and care).

Parameters

port	The port's letter (A, B, C ...)
mask	Bit values of 1 mark the bits reps. pins to be set to input

See also

inpPins

2.28.2.18 #define dirPort(*port*)

Direction port (by letter)

This low level helper would / should hardly be called directly by user / application software.

Parameters

port	The port's letter (A, B, C ...)

Returns

this is I/O-Register DDRx

See also

inpPins

2.28.3 Function Documentation

2.28.3.1 void initSystemRes (void)

Initialise system resources after reset or restart.

This function essentially calls all other initialisation functions in appropriate order (while interrupts are disabled). At return interrupts are enabled.

This function shall never be called directly by user software. To cause a software restart use systemAbort(uint8_t cause).

2.28.3.2 void initStatusLeds (uint8_t *LEDgnStart*, uint8_t *LEDrdStart*)

Initialisation of status / test LEDs (if any)

Parameters

LEDgnStart	start value for the green LED (ON or OFF)
LEDrdStart	start value for the red LED (ON or OFF)

2.28.3.3 int main (void)

The system start.

This is the reset entry point (where it all begins). This function is weAutSys' system software. It must not be supplied by user / application software.

User / application software is written by supplying one or more cyclic or event driven threads.

The system start.

This function is the bootloader's program entry. It contains the full (protocol AVR109) state machine.

Returns

main returns nothing on an embedded µController; declaring an int function is just required by C tradition

2.28.3.4 void setAbortCommand (uint8_t *command*)

Set the abort cause respectively command.

This function will set the the abort command or cause to the parameter value as long as the current state is `PLCstop` or `PLCrun`.

In other words: From state run or stop any state can be set including an abort / fault state. But an abort / fault state will not be left or changed by this function.

Parameters

command	the new run stop fault abort state / command

See also

abortHMI
abortBoot
PLCrun
PLCstop
WDtiOut

2.28.4 Variable Documentation

2.28.4.1 uint8_t resetCauses

The last reset cause(s)

This is the state of the MCU Status Register read very early at system restart.

The register will be reset after setting this variable.

This variable must not be modified elsewhere.

2.28.4.2 char const∗ resetCauseText

The last main cause as text.

This is a text describing the main reset cause according to `resetCauses`. It will be set at system start.

Not to be modified elsewhere.

The text will be at least 11 characters long including at least one trailing blank. It is in flash memory (program space). In case of full bootloader integration it will be a 32 bit address value.

Examples:

main.c.

2.28.4.3 uint8_t abortCommand

The abort command respectively cause.

Values (in Hex) assigned are:

00 : 0perate, run n0rmally

B0 : Block 0peration, stop (no application threads scheduled, PLCstop) ED : unExpected watchDog timeout

EC : Exit by abort Command

Any value not 0x00 and not 0xB0 will drive the system through a complete reset / restart procedure.

If 0x00 or 0xB0 this variable is set by system software if the corresponding abort conditions are diagnosed. The same can be done by user / application software if adequate.

See also

setAbortCommand

2.29 Software instrumentation

2.29.1 Overview

weAutSys' target hardware — mainly weAut_01 and comparable boards — often allow two LEDs and two Ports (test pins) to be used for monitoring software activity.

LEDs are limited to the human eye if no extra hardware adaption is made. Additionally the two LEDs in question are seldom really free to use. The red one e.g. is used by weAutSys to signal "low load voltage" (LV, "Lastspannung") on weAut_01.

On the other hand unused ports are really free for software instrumentation use. If neither UART flow control nor counter / generator nor I2C / twoWire is used weAut_01 4 ports are free and easily accessible on jumper pins.

Software activities indicated by outputs to those ports can be monitored and measured at CPU clock accuracy by relative simple and inexpensive equipment. One use case is is marking the CPU usage of application threads to demonstrate them to stay within their rightful limits.

As port output can be done in one CPU cycle this type of instrumentation may in most cases very well remain production software (in contrast to debugging instructions and calls).

Functions

- void initTestPins (uint8_t TP0start, uint8_t TP1start)

  Initialisation of the (potential) test pins.
- void setStatusLedGn (uint8_t state)

  Set the state of the green (test) LED.
- void setStatusLedRd (uint8_t state)

  Set the state of the red (test) LED.
- void setTestPin0 (uint8_t state) __attribute__((always_inline))

  Set the state of Test-Pin 0.
- void setTestPin1 (uint8_t state) __attribute__((always_inline))

  Set the state of Test-Pin 1.
- void toggleStatusLedGn (void)

  Change the state of the green (test) LED.
- void toggleStatusLedRd (void)

  Change the state of the red (test) LED.

2.29.2 Function Documentation

2.29.2.1 void **setStatusLedGn** (uint8_t *state*)

Set the state of the green (test) LED.

If no green test LED is available on the respective platform but a yellow one (Arduinos e.g.) the latter is used.

If neither green nor yellow single status / test LED is available this procedure does nothing.

Parameters

state	0 / false: turn off; != 0: turn on

Examples:

main.c.

2.29.2.2 void **toggleStatusLedGn** (void)

Change the state of the green (test) LED.

If no such test LED is available on the respective platform this procedure does nothing. Otherwise the LED's state is toggled.

See also

setStatusLedGn

Examples:

main.c.

2.29.2.3 void **setStatusLedRd** (uint8_t *state*)

Set the state of the red (test) LED.

If no red test LED is available on the respective platform, this procedure does nothing.

Parameters

state	0 / false: turn off; != 0: turn on

2.29.2.4 void **toggleStatusLedRd** (void)

Change the state of the red (test) LED.

If no red test LED is available on the respective platform, this procedure does nothing. Otherwise the LED's state is toggled.

See also

setStatusLedRd

2.29.2.5 void **initTestPins** (uint8_t *TP0start,* uint8_t *TP1start*)

Initialisation of the (potential) test pins.

This function sets Port C1 / testPin0 and Port C0 / testPin1 as output and does setTestPin0(ON) and setTestPin1(OFF).

Hint: Using the testpins 0 & 1 with the module weAut_01 inhibits the usage of Port C1 / C0 for other purposes like I²C / twowire. Testpin0 JP3pin1 (would be twoWire SDA) and Testpin1 is JP3pin4 (would be SCL).

Hint: If the hardware is not weAut_01 the test pins may be assigned to other ports and pins.

Parameters

TP0start	initial value for Testpin 0 (On or OFF)
TP1start	initial value for Testpin 1 (On or OFF)

Examples:

main.c.

2.29.2.6 void **setTestPin0** (uint8_t *state*)

Set the state of Test-Pin 0.

This is to be used to measure the behaviour resp. timing of software by digital analysers, oscilloscopes or other instruments.

Parameters

state	0 / false for off; != 0 for on

2.29.2.7 void **setTestPin1** (uint8_t *state*)

Set the state of Test-Pin 1.

This is to be used to measure the behaviour resp. timing of software by digital analysers, oscilloscopes or other instruments.

Parameters

state	0 / false for off; != 0 for on

2.30 Debugging aids

2.30.1 Overview

weAutSys' system software partly contains debugging instructions and calls that can be turned on at compile time by means of macros called DEBUG_<module> (DEBUG_NTP etc.) with values in the range 0..5. The higher the value the more debug information is usually supplied.

Undefined (or defined as 0 for clarity) means no debugging at all for that module throwing out all concerned calls and constants at compile time.

weAutSys supplies two handful functions that simplify the formatting of debug information and outputting them to the serial (UART) output. UART output allows almost any PC or laptop to be used as observation and logging instrument therefore.

User software is free to use the same (recommended) schema and — with care — those supporting functions.

As the production rate of debugging info may be quite high and UART speed and buffer space is limited, all these debugging helpers for UART output do spin waiting for enough space. This, of course, violates all rules for CPU usage.

Hence the use of those functions must really be restricted to debugging only.

And, by defining all said DEBUG_.. macros to 0 or other means, debugging has to be turned out completely for production software (quite in contrast to software instrumentation).

The formatters used by the debugging functions may, of course, be utilised directly without any restrictions.

The test / debug switches (macros) allow special code — mainly test outputs via standard / serial output — to be compiled in for development purposes. The higher the value the more verbose is the respective output. For an online productive system all test / debug switches should be off or 0.

The settings shown in this documentation might differ from the single weAut_01 module's generated code. If in doubt check the RS232 output with a terminal (emulator program).

Functions

- void bufLog2Dec_P (char const ∗src, uint16_t info1, uint16_t info2)

  Log a program space string + two 3 digit decimal numbers to buffered log output.
- void bufLog4HexBE (uint16_t const info)

  Log a four digit big endian hex number to buffered log output.
- void bufLog8HexBE (uint32_t const info)

  Log an eight digit big endian hex number + one space to buffered log output.
- void bufLogDec (uint16_t const info)

  Log a 4 digit decimal number to buffered log output.
- void bufLogDec3 (uint8_t const info)

  Log a three decimal digit number with leading zeroes to buffered log output.
- void bufLogDec_P (char const ∗src, uint16_t info)

  Log a string from program space + a 4 digit decimal number to buffered log output.
- void bufLogDecB (uint8_t const info)

  Log a two decimal digit number with leading zeroes to buffered log output.
- void bufLogDecHex_P (char const ∗src, uint16_t info1, uint32_t info2)

  Log a program space string + a 3 digit decimal + an 8 digit hex number to buffered log output.
- void bufLogHex (uint8_t const info)

  Log a two digit hex number to buffered log output.
- void bufLogHex_P (char const ∗src, uint8_t info)

  Log a string from program space + a two digit hex number to buffered log output.
- void bufLogHexHex_P (char const ∗src, uint8_t info1, uint32_t info2)

Log a program space string + a 2 digit and an 8 digit hex number to buffered log output.
- void bufLogLF (uint8_t n)

  Log space and linefeed to buffered log output.
- void bufLogMark (char ∗info, uint8_t const len)

  Output a (debug) marker text.
- void bufLogTThex_P (char const ∗tx1, char const ∗tx2, uint8_t info)

  Log two flash strings + a two digit hex number to buffered log output.
- void bufLogTxt (char ∗src, uint8_t n)

  Log some characters from RAM to buffered log output.
- void bufLogTxt_P (char const ∗src)

  Log some characters from program space to buffered log output.

2.30.2 Function Documentation

2.30.2.1 void **bufLogTxt_P** (char const ∗ *src*)

Log some characters from program space to buffered log output.

This debugging function does essentially the same as bufLogPutSt_P but it will not drop anything if not enough output buffer space is available. Instead it will just overwrite the oldest output.

Anyway the number of characters output will be restricted to BUF_STREAMS_CAP. And, of course, the output stops at the first 0 in `src`.

Parameters

src	the characters to be output in flash memory (0-terminated)

See also

bufLogDec_P
bufLogHex_P

2.30.2.2 void **bufLogTxt** (char ∗ *src*, uint8_t *n*)

Log some characters from RAM to buffered log output.

This debugging function does the same as bufLogTxt_P except that the string to be output is in RAM and the extra length limit `n` . Anyway the output will stop at the first 0 (zero) character.

All the restrictions and warnings are the same as for bufLogTxt_P .

Parameters

src	the characters to be output in RAM
n	the maximum number of characters to be output

See also

bufLogTxt_P
bufLogHex_P

Examples:

main.c.

2.30.2.3 void **bufLogDec_P** (char const * *src,* uint16_t *info*)

Log a string from program space + a 4 digit decimal number to buffered log output.

This debugging function outputs the flash memory text `src` and appends directly a four digit decimal number /c info right aligned.

The same restrictions and warnings apply as for bufLogTxt_P.

Parameters

src	the characters to be output in Flash memory (not NULL !)
info	the four digit number as extra info (right aligned)

See also

bufLogTxt_P
bufLogHex_P
bufLog2Dec_P

2.30.2.4 void **bufLog2Dec_P** (char const * *src,* uint16_t *info1,* uint16_t *info2*)

Log a program space string + two 3 digit decimal numbers to buffered log output.

This debugging function is similar to bufLogDec_P. It outputs the flash memory text (src, n) appends directly two three digit decimal numbers `info1` and `info2` with leading zeroes + a line feed.

The same restrictions and warnings apply as for bufLogTxt_P.

Parameters

src	the characters to be output in Flash memory
info1	the three digit number as extra info (right aligned, 0..999)
info2	the second three digit number, 0..999

See also

bufLogTxt_P
bufLogHex_P
bufLogDec_P

2.30.2.5 void **bufLogDecHex_P** (char const * *src,* uint16_t *info1,* uint32_t *info2*)

Log a program space string + a 3 digit decimal + an 8 digit hex number to buffered log output.

This debugging function is similar to bufLogDec_P. It outputs the flash memory text `src`) appends directly a three digit decimal number `info1` with leading zeroes and an 8 digit hex number `info2`

- a line feed.

The same restrictions and warnings apply as for bufLogTxt_P.

Parameters

src	the characters to be output in Flash memory
info1	the three digit number as extra info (right aligned 0..999)
info2	the eight digit hex number

See also

bufLogTxt_P
bufLogHex_P
bufLogDec_P

2.30.2.6 void **bufLogHexHex_P** (char const ∗ *src,* uint8_t *info1,* uint32_t *info2*)

Log a program space string + a 2 digit and an 8 digit hex number to buffered log output.

This debugging function is similar to bufLogDec_P. It outputs the flash memory text `src` appends directly a two digit haxadecimal number `info1`, a blank and an 8 digit hex number `info2`

- a line feed.

The same restrictions and warnings apply as for bufLogTxt_P.

Parameters

src	the characters to be output in Flash memory
info1	the two digit hex number as extra info
info2	the eight digit hex number

See also

bufLogTxt_P
bufLogHex_P
bufLogDec_P

2.30.2.7 void **bufLogHex_P** (char const ∗ *src,* uint8_t *info*)

Log a string from program space + a two digit hex number to buffered log output.

This debugging function outputs the flash memory text `src` and appends directly a two digit hexadecimal number `info`.

The same restrictions and warnings apply as for bufLogTxt_P.

Parameters

src	the characters to be output in Flash memory
info	the hex number (two digits)

See also

bufLogTxt_P
bufLogDec_P

2.30.2.8 void **bufLogTThex_P** (char const ∗ *tx1,* char const ∗ *tx2,* uint8_t *info*)

Log two flash strings + a two digit hex number to buffered log output.

This debugging function outputs the flash memory texts `tx1` and `tx2` directly concatenated. After a space the two digit hexadecimal number `info` + a line feed is output.

The same restrictions and warnings apply as for bufLogTxt_P.

Parameters

tx1	the characters to be output from flash memory (not NULL !)
tx2	more characters to be output from flash memory (not NULL !)
info	the hex number (two digits)

See also

bufLogTxt_P
bufLogDec_P

2.30.2.9 void **bufLogDec** (uint16_t const *info*)

Log a 4 digit decimal number to buffered log output.

This debugging function outputs a four digit decimal number /c info right aligned + a space.

Parameters

info	the four digit number

See also

bufLogTxt_P
bufLogHex_P
bufLog2Dec_P
bufLogDecB
bufLogDec3

2.30.2.10 void **bufLogDec3** (uint8_t const *info*)

Log a three decimal digit number with leading zeroes to buffered log output.

This debugging function outputs a three digit decimal number /c info with leading zeros ("000" .. "255").

Parameters

info	the three digit number

See also

bufLogDec
bufLogDecB

2.30.2.11 void **bufLogDecB** (uint8_t const *info*)

Log a two decimal digit number with leading zeroes to buffered log output.

This debugging function outputs a two digit decimal number /c info with leading zero. A value 0..99 will be output as two characters " 0" .. "99". Values of 100 and above will get " ∗".

Parameters

info	the two digit number

See also

bufLogDec
bufLogDec3

2.30.2.12 void **bufLogHex** (uint8_t const *info*)

Log a two digit hex number to buffered log output.

This debugging function logs a two digit hexadecimal number.

Parameters

info	the hex number (two digits)

See also

bufLogTxt_P
bufLogDec_P

2.30.2.13 void **bufLog4HexBE** (uint16_t const *info*)

Log a four digit big endian hex number to buffered log output.

This debugging function logs a 4 digits two big endian bytes hexadecimal number.

Parameters

info	the hex number (4 digits, big endian)

2.30.2.14 void **bufLog8HexBE** (uint32_t const *info*)

Log an eight digit big endian hex number + one space to buffered log output.

The same restrictions and warnings apply as for bufLogTxt_P.

Parameters

info	the hex number (8 digits, big endian)

2.30.2.15 void **bufLogMark** (char $*$ *info*, uint8_t const *len*)

Output a (debug) marker text.

The text output is " # # @ <ms-stamp>/<12,8 µs> : <info>"

Both time stamps are hexadecimal. The first one is the system run time in ms since start / reset. The second is a 12,8µs tick counter that wraps independently about every 3,27 ms.

Parameters

info	marker text (short !, might be NULL)
len	info's length (0..31)

See also

cnt12u8_8

2.30.2.16 void **bufLogLF** (uint8_t *n*)

Log space and linefeed to buffered log output.

This debugging function outputs a space + a linefeed at least once to max. 16 times if n is in the range 2..16.

The same restrictions and warnings apply as for bufLogTxt_P.

Parameters

n	the number of lines

See also

bufLogTxt_P
bufLogDec_P

2.31 System initialisation

2.31.1 Overview

These functions are used by weAutSys in due sequence to initialise the basic modules of the target, i.e mainly weAut_01, hardware. The use of those initialisation functions by user / application software requires extra care.

Data Structures

- struct conf_data_t

  The device's basic configuration data type.

Defines

- #define B1D6MODE_NONE 0

  Usage mode of ports PD6 and PB1: none.
- #define B1D6MODE_RTSCTS_FLOWC 0x11

  Usage mode of ports PD6 and PB1: RTS CTS input flow control.
- #define B1MODE_MASK 0xF0

  Usage mask for PB1.
- #define B1MODE_RTS_FLOWC 0x10

  Usage mode of port PB1 : RTS output flow control.
- #define D6MODE_CTS_FLOWC 0x01

  Usage mode of port PD6 : CTS input flow control.
- #define D6MODE_MASK 0x0F

  Usage mask for PD6.
- #define ENCBUF_END 0x1FFF

  ENC's dual port RAM end.
- #define isCTSflowC

  Usage mode of port PD6 : CTS input flow control.
- #define isRTSflowC

  Usage mode of port PB1: RTS output flow control.
- #define RXBUF_END 0x1A0D

  Receive buffer end.
- #define TXBUF_STRT 0x1A0E

  Transmit buffer start.

Functions

- void encInit (void)

  Initialise the Ethernet controller ENC28J60.
- void encReset (void)

  Reset the Ethernet controller ENC28J60.
- void encSetBufferLimits (void)

  Set the sizes of ENC28J60's receive and transmit buffers.
- uint8_t getB1D6mode (void) __attribute__((always_inline))

  Get the usage of the ports PD6 and PB1.
- void lanComStdInit (void)

  Initialise the LAN communication (standard way)
- void networkInit (void)

Initialise the network stack.
- void persistInit (void)

  Initialise persistence / EEPROM handling.
- void setB1D6mode (uint8_t mode)

  Set the usage of the ports PD6 and PB1.
- uint32_t uartBaud (uint16_t uartPrescal, uint8_t x2)

  Calculate the true baudrate for a prescaler setting.
- void uartInit (uint16_t baudDivide, uint8_t x2, uint8_t len, uint8_t parity, uint8_t stopBits)

  Initialise the serial input (UART0)
- uint16_t uartPrescaler (uint32_t baudRate, uint8_t x2)

  Calculate the prescaler setting for a desired baudrate.
- void uartSetBaudDivide (uint16_t baudDivide) __attribute__((always_inline))

  Set the UART0's baudrate divisor.

Variables

- uint16_t actPackInd

  Index of the actual receive packet in ENC's dual port memory.
- conf_data_t defaultConfData

  The device's basic default configuration data.
- conf_data_t defaultTypeConfData

  The device type's basic default configuration data in flash memory.

2.31.2 Define Documentation

2.31.2.1 #define **RXBUF_END** 0x1A0D

Receive buffer end.

This should be an odd value due to ENC's bug B7 list point 11 (called ERXND there). Due to a bug the receive buffer start will always be 0x0000 i.e. the start of ENC's dual port RAM.

This setting (1A0D = 6669) leaves 5F2 = 1522 bytes for send buffer (and all else). This in-balance is due to not having multiple send packets in the ENC buffer which would hardly be organised with uIP while having multiple incoming packets buffered is normal operation.

2.31.2.2 #define **TXBUF_STRT** 0x1A0E

Transmit buffer start.

This is the minimal and the initial / default value for the start of a transmit package in the dual port RAM. It should be even (due to an ENC bug) and must be larger than RXBUF_END as well as (considerably) smaller than the ENCBUF_END.

2.31.2.3 #define **B1D6MODE_NONE** 0

Usage mode of ports PD6 and PB1: none.

Application software may set and use these ports freely.

2.31.2.4 #define **B1MODE_MASK** 0xF0

Usage mask for PB1.

The upper four bits of b1d6Mode concern Port B1. If non of those bits is set the application software may set and use this port freely — i.e. according to hardware / jumpers.

2.31.2.5 #define D6MODE_MASK 0x0F

Usage mask for PD6.

The lower four bits of b1d6Mode concern Port D6. If non of those bits is set the application software may set and use this port freely — i.e. according to hardware / jumpers.

2.31.2.6 #define D6MODE_CTS_FLOWC 0x01

Usage mode of port PD6 : CTS input flow control.

The port PD6 must be connected to the CTS output via a level shifter / inverter (MAX202 e.g.).

In this mode the UART software will signal their ability to receive more bytes according to UART_IN_SPACE_LIM (off) resp. (3 $*$ UART_IN_SPACE_LIM) (om).

2.31.2.7 #define isCTSflowC

Usage mode of port PD6 : CTS input flow control.

See also

D6MODE_CTS_FLOWC
getB1D6mode()

2.31.2.8 #define B1MODE_RTS_FLOWC 0x10

Usage mode of port PB1 : RTS output flow control.

The port PB1 must be connected to the RTS input via a level shifter / inverter (MAX202 e.g.).

In this mode the UART software will send only if the other station sets RTS to logic 0 (+12V).

Note

This is a non-standard but nevertheless (only) reasonable symmetric usage of RTS / CTS. CTS is set here independent of RTS and not as response to it. CTS, is set by D6MODE_CTS_FLOWC to be used just indicates permission this device to send data to it. RTS indicates permission from the other station to this device to send. Hence it would better be called RTR (Ready to Receive) in the other stations point of view.

Two (weAut_01 / weAutSys) devices with both set and jumpered to B1D6MODE_RTSCTS_FLOWC may very well be P2P connected by serial link.

2.31.2.9 #define isRTSflowC

Usage mode of port PB1: RTS output flow control.

See also

B1MODE_RTS_FLOWC
getB1D6mode()

2.31.2.10 #define B1D6MODE_RTSCTS_FLOWC 0x11

Usage mode of ports PD6 and PB1: RTS CTS input flow control.

The port PD6 must be connected to the CTS output and PB1 to the RTS input — both via level shifters / inverters (MAX202 e.g.).

In this mode the UART software will use those ports for input and output flow control and all other settings and uses by application software are forbidden interference.

This is, of course, just D6MODE_CTS_FLOWC and B1MODE_RTS_FLOWC combined

2.31.3 Function Documentation

2.31.3.1 void **encSetBufferLimits** (void)

Set the sizes of ENC28J60's receive and transmit buffers.

The ENC28J60's total buffer is 8191 bytes. This function divides this to a larger buffer for received packets at the begin and a smaller part as buffer for a transmit package (and maybe for all else).

Before changing the buffer limits this function disables and resets the receiver and transmitter and leaves it so (i.e. in reset state).

The receive packet index actPackInd as well as read and write pointers are set accordingly.

/note This function uses macros (defines) to set the buffer sizes.

2.31.3.2 void **encReset** (void)

Reset the Ethernet controller ENC28J60.

This function sends a (software) reset command to the ENC28J60. Due to an ENC28J60 bug (B7 errata point 1) this reset is quite hard as it scrambles the ENC's internal clocking. The only workaround is waiting 1 ms before attempting further accesses.

This function does nothing else: a) no further ENC settings due to the bug and b) no (conditional) log message, as for usages early after reset, no logging resources (like UART e.g.) may be available.

2.31.3.3 void **encInit** (void)

Initialise the Ethernet controller ENC28J60.

This function initialises the Ethernet driver ENC28J60 but does not reset it. This function should be called not earlier than 1ms after resetting the ENC by hardware signal or software command.

This function checks ENC's (PHY) clock ready and does nothing but setting the bit ENC_HWP in networkNotReady.

Otherwise the initialisation is done and all ENC related bits in networkNotReady will be cleared.

2.31.3.4 void **networkInit** (void)

Initialise the network stack.

This function initialises the Ethernet driver ENC28J60 and the uIP stack.

The driver IC ENC28J60 is not reset (encReset) here.

See also

encInit()
lanComStdInit()

2.31.3.5 void **lanComStdInit** (void)

Initialise the LAN communication (standard way)

This function initialises the LAN communication in a standard way. If use DHCP is set in curIpConf (.useFlags) DHCP is initialiesd and nothing else is done.

Otherwise the IP configuration is set by the default configuration and NTP is initialised if be NTP client is set.

2.31.3.6 void **persistInit** (void)

Initialise persistence / EEPROM handling.

This function uses spin waiting for EEPROM (write) operations. It is to be used in initialisation (after reset) only.

2.31.3.7 void **uartSetBaudDivide** (uint16_t *baudDivide*)

Set the UART0's baudrate divisor.

This function sets the baud rate prescaler register and hence the baud rate.

It is normally called indirectly while initialising. Calling while running may spoil ongoing transmissions.

Parameters

baudDivide	0..4096

See also

uartInit

2.31.3.8 void **uartInit** (uint16_t *baudDivide,* uint8_t *x2,* uint8_t *len,* uint8_t *parity,* uint8_t *stopBits*)

Initialise the serial input (UART0)

This function is normally used after system reset.

To call this function while the system is running is not recommended or should at least be considered and tested very carefully.

UART 0 is initialised in asynchronous mode for receiving and transmitting (full duplex).

Parameters

baudDivide	0..4096 prescaler
x2	true: double speed (half receive samples)
len	5,6,7, 8 [default] (9 [not yet supported])
parity	0=false: none; odd , even value: odd, even
stopBits	2: 2 stop bits, 1 [default]: 1 stop bit

See also

uartSetBaudDivide

2.31.3.9 uint8_t **getB1D6mode** (void)

Get the usage of the ports PD6 and PB1.

These µController ports can be used in a variety of ways independent of the serial communication link. In the weAut_01 module PD5 can be jumpered to the CTS output via MAX202 level shifter / inverter and PB1 can be jumpered to the RTS input.

If so connected they may be used for input (RTS-) CTS- flow control if this mode is set so.

See also

setB1D6mode

2.31.3.10 void **setB1D6mode** (uint8_t *mode*)

Set the usage of the ports PD6 and PB1.

See also

getB1D6mode
B1D6MODE_RSTCTS_FLOWC
B1D6MODE_NONE

Parameters

mode	the new mode

See also

setUARTflowcontrolByChar

2.31.3.11 uint16_t **uartPrescaler** (uint32_t *baudRate,* uint8_t *x2*)

Calculate the prescaler setting for a desired baudrate.

The calculated and returned `prescaler` divisor setting must be in the range 0 .. 4096. Otherwise the desired `baudRate` can't even be approximated with the given processor frequency (e.g. F_CPU=20000000).

The effective baudRate can be calculated by uartBaud(uint16_t, uint8_t) to check the tolerance/error.

Note

This function probably involves 32 bit arithmetic making it a bit expensive with respect to processor usage. It is intended for use in the reset / initialisation phase. Instead of calling it regularly with a limited set of parameter values, it is recommendable to cache those repeatedly needed results.

Parameters

x2	true: double speed (half receive samples)
baudRate	the rate desired

Returns

the UART prescaler setting; it must be 0..4096

2.31.3.12 uint32_t **uartBaud** (uint16_t *uartPrescal,* uint8_t *x2*)

Calculate the true baudrate for a prescaler setting.

Note

As this function involves 32 bit arithmetic, it is expensive with respect to processor usage. It is intended for use in the reset / initialisation phase. Instead of calling it regularly with a limited set of parameter values, it's recommendable to store results calculated while initialising for later use.

Parameters

uartPrescal	0..4096 prescaler
x2	true: double speed (half receive samples)

Returns

the baudrate

See also

uartInit

2.31.4 Variable Documentation

2.31.4.1 uint16_t actPackInd

Index of the actual receive packet in ENC's dual port memory.

If at least one package has been received this will be the index to its begin. Most of the time this should be in accordance with ERXRDPT. Under most circumstances this must not be modified by user / application software.

2.31.4.2 conf_data_t defaultConfData

The device's basic default configuration data.

These are the basic configuration data specific to the individual device. To survive power off one copy of these data is to be held in EEPROM at address eeConfigAdd = EEPROM[EEPROM_POINTER2_EE_CONF].

2.31.4.3 conf_data_t defaultTypeConfData

The device type's basic default configuration data in flash memory.

These are the basic default configuration data specific to the device type. These values typically are used only as long as no other device specific (individual) values were set in persistent EEPROM storage.

So this is just the first startup default before commissioning to use.

Examples:

individEEP.c.

2.32 Basic I/O drivers (SPI)

2.32.1 Overview

weAutSys provides the basic the basic driver functions for the (embedded) periphery attached to one of the two weAut_01 SPI interfaces.

Files

- file spi.h

    weAutSys'/weAut_01' system calls and services for the SPI interfaces

Defines

- #define memCardIsSelected()

    The memory card (inserted) is chip-selected.
- #define nicIsSelected()

    The Ethernet driver is chip-selected.
- #define SPI100kHz

    control value to set SPI 2 clock to 100 kHz
- #define SPI10MHz 0

    Divisor register for UART as SPI (UBRR) setting for 10 MHz.
- #define SPI1MHz

    control value to set SPI 2 clock to 1 MHz
- #define SPI200kHz

    control value to set SPI 2 clock to 200 kHz
- #define SPI2COND_ERR_RET(cond, err, ret)

    Check a SPI2 condition with timeout.
- #define SPI2MHz

    UBRR value to set SPI clock to 2 MHz.
- #define SPI400kHz

    control value to set SPI 2 clock to 400 kHz
- #define SPI5MHz 1

    Divisor register for UART as SPI (UBRR)) setting for 5 MHz.
- #define UCPHA1

    UCSR1C register (bit number)

Functions

- uint8_t deSelectSPI1 (void) __attribute__((always_inline))

    De-select all SPI1 devices.
- void deSelectSPI1_impl (void) __attribute__((always_inline))

    De-select all SPI1 devices (raw)
- void deSelectSPI2 (void) __attribute__((always_inline))

    De-select all SPI2 devices.
- void selectDIdisplayLEDs (void) __attribute__((always_inline))

    Chip-select a SPI 1 device: the digital input display LEDS (HMI)
- void selectDOdrivDIleds (void) __attribute__((always_inline))

    Chip-select two SPI 1 devices: the DO driver and the DI LEDs (HMI)
- void selectDOdriver (void) __attribute__((always_inline))

Chip-select a SPI 1 device: the digital output driver (DO)
- void selectEtherChip (void) __attribute__((always_inline))

  Chip-select a SPI 2 device: the Ethernet driver chip.
- void selectMemCard (void) __attribute__((always_inline))

  Chip-select a SPI 2 device: the memory card (holder)
- uint8_t spi2Receive (void) __attribute__((always_inline))

  Receive one byte from SPI 2.
- void spi2ReceiveNS (uint8_t ∗receiveB, uint16_t n, uint8_t skip)

  Receive n bytes to a buffer via SPI 2 with optional skip (afterwards)
- void spi2ReceiveSN (uint8_t ∗receiveB, uint16_t skip, uint16_t n)

  Receive n bytes to a buffer via SPI 2 with optional skip (before)
- uint8_t spi2Tranceive (uint8_t sendB) __attribute__((always_inline))

  Send and receive one byte via SPI 2.
- uint8_t spi2Tranceive2 (uint8_t sendB1, uint8_t sendB2)

  Send two bytes and receive one byte via SPI 2.
- uint8_t spi2TranceiveN (uint8_t sendB, uint8_t n)

  Send a byte n times and receive one byte via SPI 2.
- void spi2Transmit (uint8_t sendB) __attribute__((always_inline))

  Send one byte over SPI 2.

Variables

- uint8_t spi2Errors

  Accumulated errors at SPI 2.
- uint8_t spi2EtherChipBaud

  The SPI (2) clock frequency used for the Ethernet driver chip.
- uint8_t spi2MemCardBaud

  The SPI (2) clock frequency used for the memory card (holder)

2.32.2 Define Documentation

2.32.2.1 #define nicIsSelected( )

The Ethernet driver is chip-selected.

This evaluates to true if the ENB28J60's / NIC's chip select is active.

See also

selectEtherChip

2.32.2.2 #define memCardIsSelected( )

The memory card (inserted) is chip-selected.

This evaluates to true if the memory card chip select is active.

See also

selectMemCard

2.32.2.3 #define SPI10MHz 0

Divisor register for UART as SPI (UBRR) setting for 10 MHz.

This value gives the highest possible (UART as) SPI baud rate, that is CPU clock / 2. That would be 10 MHz on a 20 MHz clocked µController, as for example an ATmega1284 at max. speed. An ATmega 2560 would yield max. 8 MHz with this setting.

In the same sense SPI5MHz gives corresponding results of 5 vs. 4 MHz. SPI2MHz, SPI2MHz, SPI400kHz, SPI200kHz and SPI100kHz on the other hand give their nominal value at both 20 MHz and 16 MHz CPU clock; that is on ref intro_secH "weAut_01", ArduinoMega2560, ArduinoUno and others.

2.32.2.4 #define SPI5MHz 1

Divisor register for UART as SPI (UBRR)) setting for 5 MHz.

This value gives the second highest possible (UART as) SPI baud rate, that is CPU clock / 4. That would be 5 MHz on a 20 MHz clocked µController and 4 MHz with a 16 MHz clock (ATmega1284 at max. speed e.g.). (An ATmega 2560 would yield max. 8 MHz with this setting.

See also

SPI10MHz

2.32.2.5 #define SPI2MHz

UBRR value to set SPI clock to 2 MHz.

See also

SPI10MHz

2.32.2.6 #define SPI1MHz

control value to set SPI 2 clock to 1 MHz

See also

SPI10MHz

2.32.2.7 #define SPI400kHz

control value to set SPI 2 clock to 400 kHz

See also

SPI10MHz

2.32.2.8 #define SPI200kHz

control value to set SPI 2 clock to 200 kHz

See also

SPI10MHz

2.32.2.9 #define SPI100kHz

control value to set SPI 2 clock to 100 kHz

See also

SPI10MHz

2.32.2.10 #define SPI2COND_ERR_RET(*cond,* *err,* *ret*)

Check a SPI2 condition with timeout.

This helper macro replaces the endless spin wait for SPI 2 conditions. It may be used without trailing semicolon. Example:

```
SPI2COND_ERR_RET(UCSR1A & (1<<UDRE1), 0x82, ) // wait with timeout
//  while ( !( UCSR1A & (1<<UDRE1))); // spin wait empty transmit buffer
```

Compared to the (endless) while wait the use of this macro does not add to runtime if the condition is met from start.

The maximum waiting time or timeout is $2 * (\text{UBRR1} + 1) * 10$ clocks. That is ∼20% longer than minimal necessary — but far better than endless waiting leading to watchdog reset.

Parameters

cond	conditional expression: true if ready to proceed
err	error bits to be set in spi2Errors in case of waiting for `cond` times out (by more than The xyzIsSelected bits are set by this macro
ret	return value / expression for return statement in error case; leave blank for use within void functions

2.32.3 Function Documentation

2.32.3.1 uint8_t deSelectSPI1 (void)

De-select all SPI1 devices.

This (low level helper) function de-activates the chip selects (sometimes aka slave selects) of all SPI 1 devices. For configurations without SPI1 devices nothing is done.

By polling SPI1's status it is checked before de-selecting if the interface is still busy. If so, it will be waited until ready. With higher transfer rates (CPU clock /2) that won't be long.

Returned is the first reading of the SPI1 status.

Note

If the SPI is set for the highest transfer rate this function should never wait (long) for ready if a tiny bit of useful work is done between start send/receive and this call. At high transfer rates this approach is far cheaper — and quicker — than utilising the SPI ready interrupt.

Returns

the first status reading; can be checked for "was waiting" and for past collisions

See also

deSelectSPI1_impl

2.32.3.2 void deSelectSPI1_impl (void)

De-select all SPI1 devices (raw)

This function does so without any checks or waits.

See also

deSelectSPI1()

2.32.3.3 void selectDIdisplayLEDs (void)

Chip-select a SPI 1 device: the digital input display LEDS (HMI)

This (low level helper) function activates the chip select of a SPI 1 device and de-selects all others at the same SPI bus. For configurations without that device nothing is done.

This select / de-select is done without any checks or waits.

See also

deSelectSPI1()

2.32.3.4 void selectDOdrivDIleds (void)

Chip-select two SPI 1 devices: the DO driver and the DI LEDs (HMI)

After this "double select" the same value can be output to both devices in one SPI (shift) operation.

So far this is only used for initialising both with 0 by system (start) software.

See also

selectDIdisplayLEDs()

2.32.3.5 void selectDOdriver (void)

Chip-select a SPI 1 device: the digital output driver (DO)

See also

selectDIdisplayLEDs()

2.32.3.6 void deSelectSPI2 (void)

De-select all SPI2 devices.

This (low level helper) function de-activates the chip selects (sometimes aka slave selects) of all SPI 2 devices. For configurations without such devices nothing is done.

By polling ISP2's status, it is checked before de-selecting, if the interface's transmitter is still busy. If so, it will be waited until ready, i.e. for transmission complete (TXC). If this waiting times out the appropriate error bits are set.

Note

If the SPI is set for a high transfer rate this function should never wait long for ready if a tiny bit of useful work is done between the start of send/receive and this call. At high transfer rates this "do something and than wait a bit" approach is far cheaper — and quicker — than utilising a SPI ready interrupt.

See also

deSelectSPI2_impl()

2.32.3.7 void **selectEtherChip** (void)

Chip-select a SPI 2 device: the Ethernet driver chip.

This (low level helper) function activates the chip select of a SPI 2 device and de-selects all others at the same SPI bus. For configurations without that device nothing is done.

This select / de-select is done without any checks or waits.

See also

deSelectSPI2()

2.32.3.8 void **selectMemCard** (void)

Chip-select a SPI 2 device: the memory card (holder)

This (low level helper) function activates the chip select of a SPI 2 device and de-selects all others at the same SPI bus. For configurations without that device nothing is done.

This select / de-select is done without any checks or waits.

See also

deSelectSPI2()

2.32.3.9 void **spi2Transmit** (uint8_t *sendB*)

Send one byte over SPI 2.

This function implies that the SPI device has been selected correctly before. It waits for empty transmit buffer, writes the byte `sendB` and clears the transmit complete flag.

If waiting for empty transmit buffer takes too long this function will set the appropriate error bits and does nothing else.

Parameters

sendB	the byte to be sent

See also

spi2Receive

2.32.3.10 uint8_t **spi2Receive** (void)

Receive one byte from SPI 2.

This function implies that at least one byte has been sent over the SPI to trigger or clock the receive shift and the respective byte has not yet been consumed. The function waits for the receive complete flag and than return the byte read by SPI as result.

Note

If said precondition is not met the wait will time out and this function will set the appropriate error bits and return 0.

See also

spi2Transmit
spi2Tranceive
spi2Tranceive2

Returns

the byte read from SPI (0 may but must not indicate an error

2.32.3.11 uint8_t **spi2Tranceive** (uint8_t *sendB*)

Send and receive one byte via SPI 2.

This function implies that the SPI device has been is selected correctly before.

It effectively combines spi2Transmit(sendB) and return spi2Receive() and guarantees the correct pairing of SPI transmit and receive.

On the other hand it does neither allow useful work instead of the second (TXC) wait nor utilise the SPI2 (USAT2 as SPI) double buffering.

Parameters

sendB	the byte to be sent

Returns

the byte read from SPI (0 may but must not indicate an error)

See also

spi2Transmit
spi2Receive
spi2Tranceive2

2.32.3.12 uint8_t **spi2Tranceive2** (uint8_t *sendB1*, uint8_t *sendB2*)

Send two bytes and receive one byte via SPI 2.

This function acts as if calling spi2Tranceive twice (returning the second call's result). Compared to that proceeding this function is 20% faster by utilising the SPI2's (USAT2 as SPI) double buffering. The device sees a continuous stream of 16 clocks instead of 2 times 8 clocks with a gap.

Parameters

sendB1	the byte to be sent first
sendB2	the second byte to be sent

Returns

the byte read from SPI while transmitting `sendB2` (0 may but must not indicate an error)

See also

spi2Transmit
spi2Receive
spi2Tranceive
spi2TranceiveN

2.32.3.13 uint8_t spi2TranceiveN (uint8_t *sendB,* uint8_t *n*)

Send a byte n times and receive one byte via SPI 2.

This function sends the byte `sendB` one or more times via SPI 2 discarding all receptions except for the last one. Compared to doing this by spi2Tranceive(sendB) `n` times (in a loop) this function is at least 20 % faster cause of no gaps in the byte (MoSi) respectively clock (sClk) stream.

On the other hand no useful work or PT_YIELD yielding can be done while the SPI (2) interface is busy with this action. Insofar `n` should be reasonably small.

Parameters

sendB	the byte to be sent `n` times
n	number to repeat `sendB` (0 is taken as 1)

Returns

the byte read from SPI while transmitting `sendB2` (0 may but must not indicate an error)

See also

spi2Transmit
spi2Receive
spi2Tranceive
spi2Tranceive2

2.32.3.14 void spi2ReceiveSN (uint8_t ∗ *receiveB,* uint16_t *skip,* uint16_t *n*)

Receive n bytes to a buffer via SPI 2 with optional skip (before)

This function does `skip` + `n` receptions via SPI2 (putting all ones on MoSi). The (last) `n` bytes received are put into the buffer `receiveB`.

Compared to doing this by spi2Tranceive(0xFF) `skip` + `n` times this function is at least 20 % faster. On the other hand no useful work or yielding can be done while the SPI (2) interface is busy with the action. Insofar `skip` + `n` should be reasonably small.

Parameters

receiveB	pointer to the buffer to receive `n` bytes to (must not be NULL if `n` != 0)
skip	if $>$ 0 `skip` bytes are received and forgotten before `receiveB` will be filled
n	number of bytes to be received and filled into `receiveB`

2.32.3.15 void spi2ReceiveNS (uint8_t ∗ *receiveB,* uint16_t *n,* uint8_t *skip*)

Receive n bytes to a buffer via SPI 2 with optional skip (afterwards)

This function does `skip` + `n` receptions via SPI2 (putting all ones on MoSi). The (first) `n` bytes received are put into the buffer `receiveB`.

Compared to doing this by spi2Tranceive(0xFF) `n + skip` times this function is at least 20 % faster. On the other hand no useful work or yielding can be done while the SPI (2) interface is busy with the action. Insofar `n + skip` should be reasonably small.

Parameters

receiveB	pointer to the buffer to receive `n` bytes to (must not be NULL if `n` != 0)
n	number of bytes to be received and filled into `receiveB`
skip	if > 0 `skip` bytes are received and forgotten after `filling` `n` bytes into `receiveB`

2.32.4 Variable Documentation

2.32.4.1 uint8_t spi2EtherChipBaud

The SPI (2) clock frequency used for the Ethernet driver chip.

This is the control value used to set the SPI (2) clock frequency (aka baud rate) if used with the Ethernet driver.

Hint 1: For 20 MHz FCPU_S STR "CPU frequency" the SPI (2) frequency can be set in the range 10 MHz (value 0) down to 40 kHz (value 249).

Hint 2: Due to different and un-synchronous clocks for the CPU (20 MHz) and the Ethernet chip (ENC28J60 25 MHz) not all baudrates go well. 2 MHz proofed rock solid reliable and is the default even if 5 or 10 MHz would give more performance.

default / init: SPI2MHz

See also

SPI5MHz

2.32.4.2 uint8_t spi2MemCardBaud

The SPI (2) clock frequency used for the memory card (holder)

This is the control value used to set the SPI (2) clock frequency (aka baud rate) if used with the small memory card.

Hint 1: For 20 MHz FCPU_S STR "CPU frequency" the SPI (2) frequency can be set in the range 10 MHz (value 0) down to 40 kHz (value 249). 5 MHz is the default or reset initialisation value.

Hint 2: Most small memory cards require 400 kHz or slower during initialisation (power up and idle state). This can be done by smcSetIdle(1) no matter the spi2MemCardBaud value After initialisation and depending on the card type information read a higher frequency can and should be set (here).

See also

SPI50MHz
SPI400kHz

2.32.4.3 uint8_t spi2Errors

Accumulated errors at SPI 2.

Driver function of SPI 2 (UART1 as SPI) record seen errors in this byte. It may be reset by user or system software (especially after logging or displaying the error(s)).software.

Bit 7 (8): any error. This bit will be set on every recognised SPI 2 error. It may be reset alone leaving all others set.

Bit (4)6: any memory card error. This bit will be set on every recognised SPI 2 error if at that time the small memory card holder is selected. Bit 5 (2): any network interface / ENC28J60 error. This bit will be set on every recognised SPI 2 error if at that time the Ethernet chip is selected.

Bit 3 (8): network interface controller (NIC, ENC28J60) stays busy. This bit will be set if waiting for the MISTAT.BUSY bit to go away timed out.

Bit 2 (4): transmit complete failed. This bit will be set if waiting for transmission complete (TXC) timed out.

Bit 1 (2): transmitter buffer empty failed. This bit will be set if waiting for empty transmitter buffer (UDRE) timed out.

Bit 0 (1): receive complete failed. This bit will be set if waiting for receive complete (RXC) timed out.

2.33 Basic serial communication drivers

2.33.1 Overview

The primary target hardware for weAutSys is the automation controller weAut_01. It uses AVR's (second) UART1 as an exclusive SPI interface for the Ethernet chip. (All other SPI peripherals use the standard SPI / ISP interface.)

Thus only one UART, namely UART 0 (PD0, PD1 on ATmega 644 /1284) is available. It is handled by the driver software defined here in a way suitable for stdin, stdout and stderr as described in stdio.h. As there is only one UART handled here, just `uart` resp. `UART` is used as part of names, without any numbering like `uart0`, `uart1` etc.

As weAutSys' system and user software is organised as cooperating protothreads no blocking is allowed in called (library) functions. (This is one of Protothreads' basic characteristics.)

So no input function defined / implemented here may block in any way. This especially holds for those (indirectly) called via stdio functions. Hence a character / byte input function must immediately return a valid code or an error respectively EOF.

The same is true for output. It must in no way hang on full buffers. Even in that situation (that can easily be avoided) it has to return discarding the output.

For suitable I/O functions see module Serial Communication.

Defines

- #define FROM_BUFFER2UART()

    Driver helper.
- #define SER_RECVBUF_EMPTY

    The serial input buffer is empty.
- #define SER_RECVBUF_NOT_EMPTY

    The serial input buffer is not empty.
- #define SER_SENDBUF_EMPTY

    The serial output buffer is empty.
- #define SER_SENDBUF_NOT_EMPTY

    The serial output buffer is not empty.
- #define UART_CAN_SEND

    UART (0) can get send data.

Functions

- uint8_t uartClearOutBuffer (void)

    Clear the internal buffer for serial output.

Variables

- uint8_t uartInBuf []

    The serial input buffer.
- uint8_t uartInBufRi

    The serial input buffer read/get index.
- uint8_t uartInBufWi

    The serial input buffer write/put index.
- uint8_t uartOutBuf []

    The serial output buffer.
- volatile uint8_t uartOutBufRi

The serial output buffer read/get index.
- volatile uint8_t uartOutBufWi

  The serial output buffer write/put index.

2.33.2 Define Documentation

2.33.2.1 #define FROM_BUFFER2UART( )

Driver helper.

This is a low level helper macro for driver programming.

Warning: It is to be used by system software only and not outside atomic locks.

2.33.2.2 #define UART_CAN_SEND

UART (0) can get send data.

This evaluates to non 0 (true), if the transmit data register is empty.

2.33.2.3 #define SER_SENDBUF_NOT_EMPTY

The serial output buffer is not empty.

This evaluates to true, if the serial output buffer is NOT empty.

Examples:

main.c.

2.33.2.4 #define SER_SENDBUF_EMPTY

The serial output buffer is empty.

This evaluates to true, if the serial output buffer is empty.

2.33.2.5 #define SER_RECVBUF_EMPTY

The serial input buffer is empty.

This evaluates to true, if the serial input buffer is empty.

2.33.2.6 #define SER_RECVBUF_NOT_EMPTY

The serial input buffer is not empty.

This evaluates to true, if the serial input buffer is NOT empty.

2.33.3 Function Documentation

2.33.3.1 uint8_t uartClearOutBuffer (void)

Clear the internal buffer for serial output.

This function returns the number of characters forgotten. If 0 is returned the buffer was empty anyway.

This function is to be used to recover from buffer overrun or similar errors or to "forget the past" out of any other reasons.

2.34 Basic Ethernet communications drivers

2.34.1 Overview

These driver functions are the link between

- a 28J60 Ethernet (hardware) driver chip, coupled via SPI and
- uIP [Adam Dunkels] as TCP/IP stack.

If appropriate these low level driver and helper functions can be used in the application software as well.

Files

- file enc28j60.h

  weAutSys' (weAut_01's) 28J60 Ethernet driver

Defines

- #define BFC_CMD

  Bit field clear command.
- #define BFS_CMD

  Bit field set command.
- #define COLSTAT

  collision is occurring (Bit number)
- #define deSelectEnc()

  De-select the ENC28J60.
- #define DPXSTAT

  configured for full-duplex (Bit number)
- #define FRCLNK

  make link up even when no partner station detected (Bit number)
- #define HDLDIS

  half duplex loopback disable (Bit number)
- #define JABBER

  turn jabber correction off (Bit number)
- #define JBSTAT

  transmission met jabber criteria since last PHSTAT1 read
- #define LA_BFAST

  see PHLCON
- #define LA_BSLOW

  see PHLCON
- #define LA_COL

  see PHLCON
- #define LA_DSC
- #define LA_DSTAT

  see PHLCON
- #define LA_LEDOFF

  see PHLCON
- #define LA_LEDON

  see PHLCON
- #define LA_LSRX

see PHLCON
- #define LA_LSTAT

  see PHLCON
- #define LA_LSTXRX

  see PHLCON
- #define LA_RX

  see PHLCON
- #define LA_RXTX

  see PHLCON
- #define LA_TX

  see PHLCON
- #define LB_BFAST

  see PHLCON
- #define LB_BSLOW

  see PHLCON
- #define LB_COL

  see PHLCON
- #define LB_DSC

  see PHLCON
- #define LB_DSTAT

  see PHLCON
- #define LB_LEDOFF

  see PHLCON
- #define LB_LEDON

  see PHLCON
- #define LB_LSRX

  see PHLCON
- #define LB_LSTAT

  see PHLCON
- #define LB_LSTXRX

  see PHLCON
- #define LB_RX

  see PHLCON
- #define LB_RXTX

  see PHLCON
- #define LB_TX

  see PHLCON
- #define LLSTAT

  link was up continuously since last PHSTAT1 read
- #define LSTAT

  link is currently up (Bit number)
- #define NOSTRCH

  do not stretch LED events (see PHLCON)
- #define PFDPX

  full duplex capable (bit 12 is always set)
- #define PHCON1

  PHY control register 1.
- #define PHCON2

  PHY control register 2.
- #define PHDPX

  half duplex capable (bit 11 is always set)

- #define PHID1

  PHY identification register 1.
- #define PHID2

  PHY identification register 2.
- #define PHIE

  PHY interrupt enable register.
- #define PHIR

  PHY interrupt request register.
- #define PHLCON

  LED Configuration Register.
- #define PHSTAT1

  PHY status register 1.
- #define PHSTAT2

  PHY status register 2.
- #define PLRITY

  polarity of TPIN is reversed
- #define RBM_CMD

  Read buffer memory command.
- #define RCR_CMD

  Read control register command.
- #define RXSTAT

  PHY is currently receiving (Bit number)
- #define selectEnc()

  Chip select the ENC28J60.
- #define SYSRST_CMD 0xFF

  Reset ENC28J60 command.
- #define TLSTRCH

  LED pulse stretch, long 140ms (see PHLCON)
- #define TMSTRCH

  LED pulse stretch, medium 70ms (see PHLCON)
- #define TNSTRCH

  LED pulse stretch, normal 40ms (see PHLCON)
- #define tranceiveEnc(sendB)

  Transmit and receive one byte to / from ENC28J60.
- #define tranceiveEnc2(sendB1, sendB2)

  Transmit two bytes and receive one byte to / from ENC28J60.
- #define TXDIS

  disable twisted-pair transmitter hardware driver (Bit number)
- #define TXSTAT

  PHY is currently transmitting (Bit number)
- #define WBM_CMD

  Write buffer memory command.
- #define WCR_CMD

  Write control register command.

Control registers

- #define ESTAT

  ETHERNET status register.
- #define CLKRDY

  Clock (oscillator) is ready (ESTAT bit number)
- #define TXABRT

  The transmit request was aborted (ESTAT bit number)
- #define ECON2

  ETHERNET control register 2.
- #define AUTOINC

  automatic increment and wrap of RAM buffer addresses (ECON2 bit number)
- #define PKTDEC

  decrement (received) packets count (ECON2 bit number)
- #define PWRSV

  switch the PHY interface off (save power when not needed; ECON2 bit number)
- #define VRPS

  only if PWRSV set voltage regulator to low current (ECON2 bit number)
- #define ECON1

  ETHERNET control register 1.
- #define TXRST

  Transmit only reset (ECON1 bit number)
- #define RXRST

  Receive only reset (ECON1 bit number)
- #define DMAST

  DMA operation (within internal RAM) start and busy bit (ECON1 bit number)
- #define CSUMEN

  DMA hardware calculates checksums (ECON1 bit number)
- #define TXRTS

  The transmit logic is attempting to transmit a packet (ECON1 bit number)
- #define RXEN

  Receive enable.
- #define EIE

  Ethernet interrupt enable register.
- #define INTIE

  EIE bit number
- #define PKTIE

  EIE bit number
- #define DMAIE

  EIE bit number
- #define LINKIE

  EIE bit number
- #define TXIE

  EIE bit number
- #define TXERIE

  EIE bit number
- #define RXERIE

  EIE bit number
- #define EIR

  Ethernet interrupt request register.
- #define PKTIF

EIR bit number
- #define DMAIF

  EIR bit number
- #define LINKIF

  EIR bit number
- #define TXIF

  EIR bit number
- #define TXERIF

  EIR bit number
- #define RXERIF

  EIR bit number

Symbolic names for direct register access

The lower 5 bits (0..31) give the register number in each bank.

The bits 5 and 5 (mask 0x60) are used for the bank number (0..3 $*$ 32).

- uint8_t currentBank

  The currently set bank of registers.
- #define ERDPTL

  The buffer read address register.
- #define ERDPTH

  the buffer read address register
- #define EWRPTL

  The buffer write address register.
- #define EWRPTH

  the buffer write address register
- #define ETXSTL

  The write / transmit buffer range.
- #define ETXSTH

  the write buffer range
- #define ETXNDL

  the write buffer range
- #define ETXNDH

  the write buffer range
- #define ERXSTL

  The read / receive buffer range.
- #define ERXSTH

  the write buffer range
- #define ERXNDL

  the write buffer range
- #define ERXNDH

  the write buffer range
- #define ERXRDPTL

  The receive buffer (already) read address register.
- #define ERXRDPTH

  receive buffer already read
- #define ERXWRPTL

  The receive buffer write address register.
- #define ERXWRPTH

receive buffer write
- #define EDMASTL

  DMA start low byte.
- #define EDMASTH

  DMA start high byte.
- #define EDMANDL

  DMA end low byte.
- #define EDMANDH

  DMA end high byte.
- #define EDMADSTL

  DMA destination low byte.
- #define EDMADSTH

  DMA destination high byte.
- #define EDMACSL

  DMA checksum low byte.
- #define EDMACSH

  DMA checksum high byte.
- #define EHT0

  hash table byte 0
- #define EHT1

  hash table byte 1
- #define EHT2

  hash table byte 2
- #define EHT3

  hash table byte 3
- #define EHT4

  hash table byte 4
- #define EHT5

  hash table byte 5
- #define EHT6

  hash table byte 6
- #define EHT7

  hash table byte 7
- #define EPMM0

  pattern match mask byte 0
- #define EPMM1

  pattern match mask byte 1
- #define EPMM2

  pattern match mask byte 2
- #define EPMM3

  pattern match mask byte 3
- #define EPMM4

  pattern match mask byte 4
- #define EPMM5

  pattern match mask byte 5
- #define EPMM6

  pattern match mask byte 6
- #define EPMM7

  pattern match mask byte 7
- #define EPMCSL

  pattern match checksum low byte

- #define EPMCSH

  pattern match checksum high byte
- #define EPMOL

  pattern match offset low byte
- #define EPMOH

  pattern match offset high byte
- #define ERXFCON

  Ethernet receive filter control register.
- #define EPKTCNT

  Ethernet package count.
- #define MACON1

  MAC control register 1.
- #define TXPAUS

  MAC control register 1 (bit number)
- #define RXPAUS

  MAC control register 1 (bit number)
- #define PASSALL

  MAC control register 1 (bit number)
- #define MARXEN

  MAC control register 1 (bit number)
- #define MACON2

  MAC control register 2 (inofficial)
- #define MACON3

  MAC control register 3.
- #define PADCFG2

  MAC control register 3 (bit number)
- #define PADCFG1

  MAC control register 3 (bit number)
- #define PADCFG0

  MAC control register 3 (bit number)
- #define EXPSF64

  All short frames will be padded to 64 Bytes and have valid CRC appended.
- #define EXPSF60

  All short frames will be padded to 60 Bytes and have valid CRC appended.
- #define NOSFPAD

  No handling short frames.
- #define DTCTVLAN

  Automatic handling short frames.
- #define TXCRCEN

  MAC control register 3 (bit number)
- #define PHIDREN

  MAC control register 3 (bit number)
- #define HFRMEN

  MAC control register 3 (bit number)
- #define FRMLNEN

  MAC control register 3 (bit number)
- #define FULDPX

  MAC control register 3 (bit number)
- #define MACON4

  MAC control register 4.
- #define DEFER

MAC control register 4 (bit number)
- #define BPEN

  MAC control register 4 (bit number)
- #define NOBKOFF

  MAC control register 4 (bit number)
- #define MABBIPG

  back-to-back inter-packet gap
- #define MAIPGL

  non back-to-back inter-packet gap low
- #define MAIPGH

  non back-to-back inter-packet gap high
- #define MACLCON1

  retransmission maximum (4 bit)
- #define MACLCON2

  collision window (6 bit)
- #define MAMXFLL

  maximum frame length low byte
- #define MAMXFLH

  maximum frame length high byte
- #define MICMD

  MII command register.
- #define MIISCAN

  MII command register (bit number)
- #define MIIRD

  MII command register (bit number)
- #define MIREGADR

  MII register address register.
- #define MIWRL

  MII write data low byte.
- #define MIWRH

  MII write data high byte.
- #define MIRDL

  MII read data low byte.
- #define MIRDH

  MII read data high byte.
- #define MAADR1

  MAC address register (first)
- #define MAADR2

  MAC address register.
- #define MAADR3

  MAC address register.
- #define MAADR4

  MAC address register.
- #define MAADR5

  MAC address register.
- #define MAADR6

  MAC address register (last)
- #define EBSTSD

  built-in self test fill seed
- #define EBSTCON

  Ethernet self test control register.

- #define EBSTCSL

  built-in self test checksum low byte
- #define EBSTCSH

  built-in self test checksum high byte
- #define MISTAT

  MII status register.
- #define NVALID

  MII status register (bit number)
- #define SCAN

  MII status register (bit number)
- #define BUSY

  MII status register (bit number)
- #define EREVID

  Ethernet revision ID (5 bit, R/O)
- #define ECOCON

  Clock output control.
- #define EFLOCON

  EFLOCON (Ethernet Flow Control.
- #define EPAUSL

  Pause timer value low byte.
- #define EPAUSH

  Pause timer value high byte.
- #define REG_MASK 0x1F

  Mask for register number bits in symbolic register address.
- #define REG_BANK_MASK 0x60

  Mask for bank bits in symbolic register address.
- #define SPI_BANK_MASK

  Mask for bank bits in control register.
- #define KEY_REGISTERS 0x1B

  First number of common registers.

Per packet control bits for transmission settings

- #define PHUGEEN

  Huge frame enable (bit number)
- #define PPADEN

  per packet padding enable (bit number)
- #define PCRCEN

  per packet CRC enable (bit number)
- #define POVERRIDE

  packet overrides (bit number)

Functions

- void clearBitfield (uint8_t regAdd, uint8_t data)

  Clear bits in a control register.
- void encDisablePowersave (void)

  Leave the power safe mode.
- void encEnablePowersave (void)

  Go to power safe mode.

- void encGetMacAdd (eth_addr_t ∗mac)

  Read the current MAC address.
- uint8_t encSetMacAdd (eth_addr_t ∗mac)

  Set the MAC address.
- void putBufferMemory (uint8_t data)

  Write one byte to ENC28J60's memory buffer.
- void readBufferMemory (uint8_t ∗dest, uint16_t n)

  Read n bytes from ENC28J60's memory buffer.
- uint8_t readControlRegister (uint8_t regAdd)

  Read a control register.
- uint16_t readPhysicalRegister (uint8_t regAdd)

  Read one of ENC28J60's physical registers.
- uint8_t setBank (uint8_t regAdd)

  Sets a register bank (in ECON1) if necessary for the given register.
- void setBitfield (uint8_t regAdd, uint8_t data)

  Set bits in a control register.
- void writeBufferMemory (uint8_t ∗source, uint16_t n)

  Write n bytes to ENC28J60's memory buffer.
- void writeControlRegister (uint8_t regAdd, uint8_t data)

  Write a control register.
- void writePhysicalRegister (uint8_t regAdd, uint16_t data)

  Write to one of ENC28J60's physical registers.

Variables

- uint8_t receiveStatVec [ ]

  Receive status vector.
- uint8_t transmitStatVec [ ]

  Transmission status vector.

2.34.2 Define Documentation

2.34.2.1 #define **PHLCON**

LED Configuration Register.

Eleven bits (1..11) of this register control the operation of the two LEDs that may be driven by the ENC28J60. They are usually, as in the weAut_01 board, integrated in the RJ45 jack.

Bits 15 and 14 must be written as 1; bits 13, 12 and 0 must be written as 0.

Both LEDs, referred to as LEDA and LEDB, have the same set of options coded to other bit positions. Available bit field options by macros are named accordingly beginning with LA or LB:

Lx_DSC: Duplex status and collision activity (always stretched)

Lx_LSTXRX: Link status transmission and receive activity (always stretched)

Lx_LSRX: Link status receive activity (always stretched)

Lx_BSLOW: Blink slow

Lx_BFAST: Blink fast

Lx_LEDOFF: LED off

Lx_LEDON: LED on

Lx_RXTX: Receive and Transmission activity

Lx_DSTAT: Duplex status

Lx_LSTAT: Link status

Lx_COL: Collision activity (stretchable)

Lx_RX: Receive activity (stretchable)

Lx_TX: Transmit activity (stretchable)

The configuration for LED on's stretching for stretchable events is common for both LEDs.

2.34.2.2 #define LA_DSC

see PHLCON

2.34.2.3 #define RXEN

Receive enable.

Packets complying to the filter criteria set will be written into the receive buffer (ECON1 bit number).

2.34.2.4 #define ERDPTL

The buffer read address register.

The ENC28J60's 8K dual port RAM can be read using the (indirect) (13 bit) address in ERDPTH ERDPTL .

If the AUTOINC bit is set in ECON2 this register is incremented after each read automatically wrapping around within the receive buffer range ERXST .. ERXND.

2.34.2.5 #define ERDPTH

the buffer read address register

See also

ERDPTL

2.34.2.6 #define EWRPTL

The buffer write address register.

The ENC28J60's 8K dual port RAM can be written using the (indirect) (13 bit) address in EWRPTH : EWRPTL .

If the AUTOINC bit is set in ECON2 this address is incremented after each write automatically wrapping around within the RAM address range 0x0000 .. 0x1FFF .

2.34.2.7 #define EWRPTH

the buffer write address register

See also

EWRPTL

2.34.2.8 #define ETXSTL

The write / transmit buffer range.

The write buffer is located in the address range ETXSTH : ETXSTL .. ETXNDH : ETXNDL within the ENC28J60's 8K dual port RAM.

If the AUTOINC bit is set in ECON2 this is incremented after each read automatically wrapping around within range 0x0000 .. 0x1FFF i.e. the write wrapping is independent of this setting.

See also

ERXSTL

2.34.2.9 #define ETXSTH

the write buffer range

See also

ETXSTL

2.34.2.10 #define ETXNDL

the write buffer range

See also

ETXSTL

2.34.2.11 #define ETXNDH

the write buffer range

See also

ETXSTL

2.34.2.12 #define ERXSTL

The read / receive buffer range.

The read buffer is located in the address range ERXSTH : ERXSTL .. ERXNDH : ERXNDL within the ENC28J60's 8K dual port RAM.

If the AUTOINC bit is set in ECON2 this is incremented after each read automatically wrapping around within the receive buffer address range ERXST .. ERXND.

See also

ETXSTL

2.34.2.13 #define ERXSTH

the write buffer range

See also

ERXSTL

2.34.2.14 #define ERXNDL

the write buffer range

See also

ERXSTL

2.34.2.15 #define ERXNDH

the write buffer range

See also

ERXSTL

2.34.2.16 #define ERXRDPTL

The receive buffer (already) read address register.

The address ERXRDPTH : ERXRDPTL is the index in the ENC28J60's 8K dual port RAM to where received bytes were processed by the software.

This location is never overwritten by the receiver.

See also

ERXWRPTL

2.34.2.17 #define ERXRDPTH

receive buffer already read

See also

ERXRDPTL

2.34.2.18 #define ERXWRPTL

The receive buffer write address register.

The ENC28J60's 8K dual port RAM is written while receiving using the (indirect 13 bit) address in ERXWRPTH : ERXWRPTL auto-incrementing it after each received byte.

The address ERXRDPTH : ERXRDPTL will never be overwritten

2.34.2.19 #define ERXWRPTH

receive buffer write

See also

ERXWRPTL

2.34.2.20 #define EXPSF64

All short frames will be padded to 64 Bytes and have valid CRC appended.

MAC control register 3 (bit pattern)

2.34.2.21 #define EXPSF60

All short frames will be padded to 60 Bytes and have valid CRC appended.

MAC control register 3 (bit pattern)

2.34.2.22 #define NOSFPAD

No handling short frames.

MAC control register 3 (bit pattern)

2.34.2.23 #define DTCTVLAN

Automatic handling short frames.

The MAC will automatically detect VLAN Protocol frames (by 8100h type field). It pads short frames to 64 bytes if it's a valid VLAN frame or to 60 bytes otherwise. A valid CRC will then be appended.

MAC control register 3 (bit pattern)

2.34.2.24 #define MIREGADR

MII register address register.

This control register (5 bit) is used to address other register types. 16 bit registers are indirectly accessed via (this) address register and write respectively read registers.

See also

MIWRL
MIWRH
MIRDL
MIRDH

2.34.2.25 #define MIWRL

MII write data low byte.

See also

MIREGADR

2.34.2.26 #define **MIWRH**

MII write data high byte.

See also

MIREGADR

2.34.2.27 #define **MIRDL**

MII read data low byte.

See also

MIREGADR

2.34.2.28 #define **MIRDH**

MII read data high byte.

See also

MIREGADR

2.34.2.29 #define **SPI_BANK_MASK**

Mask for bank bits in control register.

The last 2 Bits of ECON1 are the bank value (0..3).

2.34.2.30 #define **KEY_REGISTERS** 0x1B

First number of common registers.

Control registers live in 4 banks of each theoretically 32 registers.

The last "key" or "common" registers appear in all for banks. For them there's no need to even consider bank switching.

2.34.2.31 #define **deSelectEnc**( )

De-select the ENC28J60.

Does the de-select after (waiting for) send (on UART as SPI 2 to) is completed.

2.34.2.32 #define **tranceiveEnc2**(*sendB1,* *sendB2*)

Transmit two bytes and receive one byte to / from ENC28J60.

Parameters

sendB1	the byte to be sent first
sendB2	the second byte to be sent

Returns

the (second) byte read from SPI while transmitting `sendB2`

2.34.3 Function Documentation

2.34.3.1 uint8_t readControlRegister (uint8_t *regAdd*)

Read a control register.

This function reads one of the ENC28J60's control registers.

The register's symbolic address consists of the register number (0..31 within the register bank) + the bank number (0..3)$*$32. All symbolic control register names — i.e. macros — within ENC28J60.h are formed in this way.

The bank is switched only if necessary.

All control registers are 8 bit. They fall into one of three categories: ETH, MAC and MII. One step bit set and bit clear operations eliminate the need of going through read/modify/write — but only for the ETH type.

For reading the MAC or MII registers the answer will be prepended by an extra or dummy byte. In other words: The ENC28J60 needs 16 SPI clocks to read from an ETH register and 24 SPI clocks to read from a MAC/MII register. This (little) complication is, of course, handled by this function.

Parameters

regAdd	the register's symbolic address

Returns

it's value

See also

writeControlRegister
setBank
setBitfield

2.34.3.2 void writeControlRegister (uint8_t *regAdd,* uint8_t *data*)

Write a control register.

This function writes one of the ENC28J60's control registers.

The bank is switched if necessary.

Parameters

regAdd	the register's symbolic address
data	the new value

See also

readControlRegister

2.34.3.3 void setBitfield (uint8_t *regAdd,* uint8_t *data*)

Set bits in a control register.

This is an OR operation on the content of the register regAdd. For ETH type control registers it is done internally by the ENC28J60 saving a read modify write cycle.

For MII and MAC registers this internal OR is not available and hence the read and if necessary modify + write is done.

Parameters

regAdd	symbolic register address
data	the bits to be set

See also

readControlRegister
clearBitfield

2.34.3.4 void **clearBitfield** (uint8_t *regAdd,* uint8_t *data*)

Clear bits in a control register.

This is an AND NOT on the content of the register regAdd. Conditions for a quick internal implementation by the ENC28J60 are the same as for setBitfield.

Parameters

regAdd	symbolic address of the register to be modified
data	the bits to be reset

See also

readControlRegister
setBitfield

2.34.3.5 uint16_t **readPhysicalRegister** (uint8_t *regAdd*)

Read one of ENC28J60's physical registers.

This function handles the indirect access to those registers. All physical registers are 16 bit and can be read or written as whole words only.

The complication of being only indirectly accessible via MIREGADR etc. is handled by this function.

Parameters

regAdd	the register address

Returns

its value

See also

writePhysicalRegister

2.34.3.6 void **writePhysicalRegister** (uint8_t *regAdd,* uint16_t *data*)

Write to one of ENC28J60's physical registers.

Parameters

regAdd	the register address
data	the (16 bit) value to be written

See also

readPhysicalRegister

2.34.3.7 void **readBufferMemory** (uint8_t ∗ *dest,* uint16_t *n*)

Read n bytes from ENC28J60's memory buffer.

This function reads a sequence of bytes from the dual port RAM indirectly addressed by the ERXRDPT control registers. For n $>$ 1 this usually makes sense only if auto-increment is on.

Parameters

dest	pointer to buffer in RAM to write to
n	number of bytes to read from ENC28J60's memory

2.34.3.8 void **putBufferMemory** (uint8_t *data*)

Write one byte to ENC28J60's memory buffer.

This function writes to the dual port RAM byte indirectly addressed by the EWRPT control registers.

Parameters

data	to be written into the buffer memory

2.34.3.9 void **writeBufferMemory** (uint8_t ∗ *source,* uint16_t *n*)

Write n bytes to ENC28J60's memory buffer.

This function writes a sequence of bytes to the dual port RAM byte indirectly addressed by the EWRPT control registers. This usually makes sense only if auto-increment is on.

Parameters

source	pointer to data in RAM to be written to ENC28J60's memory
n	number of bytes to be transferred from RAM to ENC

2.34.3.10 uint8_t **setBank** (uint8_t *regAdd*)

Sets a register bank (in ECON1) if necessary for the given register.

This function does nothing if no bank switch is needed to access the register regAdd

Parameters

regAdd	symbolic address of the register (bank bits relevant)

Returns

the lower 5 bits (the "bankless" address) of regAdd

2.34.3.11 void encGetMacAdd (eth_addr_t ∗ *mac*)

Read the current MAC address.

The MAC address will be read from the Ethernet driver (ENC28J60) and put in the structure supplied.

Parameters

mac	the (6 byte) address structure

2.34.3.12 uint8_t encSetMacAdd (eth_addr_t ∗ *mac*)

Set the MAC address.

The MAC address from the structure supplied will be written to the Ethernet driver (ENC28J60).
This initialisation function is usually called as

```
encSetMacAdd(& actMACadd);
```

using the Ethernet stack's (uIP's) set MAC address for the driver.

Both the driver's and the stack's setting must be kept the same.
The uIP's default / initialisation value for actMACadd is

```
{0x40,0x1B,0x50,0xCA,0xFE,0x01}
```

40 is "locally assigned unicast 0"

1B50 read "IBS 0" or Inbetriebsetzung (comissioning to use) 0 is a fixed preset for first test or standalone use

CAFE01 in this context means "check all functional element of weAut01

As this is one of the most basic ENC28J60 initialisations this function checks the chips and the SPI's operation by re-reading the ENC's MAC address registers. 0 is returned only if this check passes.

Note

This test was implemented because the CLKRDY bit test, that would do the same basic hardware testing, simply does not work on most ENC cip releases and hence can as well be omitted.

Parameters

mac	the (6 byte) address structure

Returns

0 : OK; 1..6 the number of failed check reads

2.34.3.13 void encEnablePowersave (void)

Go to power safe mode.

The ENC28J60 can be set to save power (if not used).

2.34.3.14 void **encDisablePowersave** (void)

Leave the power safe mode.

See also

encEnablePowersave

2.34.4 Variable Documentation

2.34.4.1 uint8_t **currentBank**

The currently set bank of registers.

Holds only the bank select bits, BSEL1:BSEL0 (mask 0x03) as in ECON1 register shifted 5 bits to left resp. $*$ 32 (mask 0x60) as in the symbolic names.

Not to be modified by user software.

2.34.4.2 uint8_t **receiveStatVec**[ ]

Receive status vector.

Holds ENC's 4 byte receive status vector belonging to the last received package.

2.34.4.3 uint8_t **transmitStatVec**[ ]

Transmission status vector.

Used to see any failures during transmission. It is generated after the ESTAT.TXABRT flag is set by the ENC28J60.

2.35 + + Bootloader support + +

2.35.1 Overview

weAutSys comes with a serial bootloader according to Atmel Corporation's application note AVR109.

- Serial means using weAut_01 V.24/RS232 link respectively ATMega1284P's UART0.
- AVR109 is Atmel's application note "Self-programming". It defines a protocol with one letter commands and binary data for reading and writing the own flash and EEPROM memory, checking fuses and some more.
- Besides being made for weAut_01 respectively for ATmega1284P the bootloader is also adapted to ArduinoMega2560 and ArduinoMegaADK respectively to ATmega2560. It can easily been adapted to other platforms resp. all ATmegas with bootloader or self-programming facilities. See file boot109.c for details.
- Cause of the separate usability the bootloader is in a separate project respectively directory that must be the the neighbor of the using projects, like this weAutSys.

For sake of transmission robustness it uses 38400 baud (38400N1). That makes the it a bit slower than a mySmart-USB light. Good conditions provided higher baud rates are feasible too.

The biggest advantage of this serial AVR109 bootloader integrated to weAutSys is no need for any extra programming hardware — not even those four jumpers for a bit banging ISP. The minus is not being able to modify some fuses or, of course, the bootloader itself. For most use cases that's more of a plus.

Please download and read

Albrecht Weinert

`A serial bootloader for ATmega based products -- weAut_01, Arduino and akin`

This development report `(English, .pdf)` gives some background on ATmega architecture, bootloader, avr-gcc, bootloader linking and on some common errors and problems. One of those problems, well known with Arduinos and serial bootloaders, is e.g. the dreaded:

"avrdude: error: programmer did not respond to command: set addr"

or "... set extAdr" and suchlike.

Modules

- Bootloader operation
- Bootloader integration

Files

- file boot109.c

  Implementation of weAutSys' serial bootloader program.
- file boot109.h

  Definitions of weAutSys' bootloader and helper functions.
- file bootLib.c

  Implementation ofweAutSys' bootloader helper functions.

2.36 Bootloader operation

2.36.1 Overview

In weAutSys it is intended that the bootloader software is entered on every reset. It may be entered by (CLI) user command out of normal operation also.

Before doing its AVR109 job and before (re-) entering the system/application software this bootloader implementation does a lot of (weAut_01) initialisation. See Bootloader integration for how to utilize that and Bootloader support for an overview and further references.

There are multiple criteria for the bootloader either to go directly to the application software or to enter the AVR109 protocol and waiting for a command or a sync (ESC) character. The most important are:

- application flash is cleared ([00] = FFFF): stay in AVR109
- entered by call / jump from application, i.e. by (CLI) command: remain in AVR109 (at least for a timeout longer than a minute)
- watchdog reset: enter system / application software after a 4 s timeout (to allow multiple avrdude executions in a script file)
- other reset while "enter" key pressed: remain in AVR109
- receive exit bootloader ('E') command: enter system / application software
- time-out while waiting for AVR109 bootloader command or sync (ESC) byte

When the bootloader has entered (and remained in) the AVR109 protocol part it may be used via the serial link by any programming software capable of this protocol, like e.g. the well known avrdude. Use it by:

```
avrdude -p atmega1284p -c avr109 -b 38400  -P com1  -v
```

Use it interactively:

```
avrdude -p atmega1284p -c avr109 -b 38400  -P com1  -v -t
```

Use it to burn an application program by:

```
avrdude -p atmega1284p -c avr109 -b 38400  -P com1  -v -U flash:w:main.hex
```

Should this AVR bootloder mode be entered while a person's terminal is serially connected (e.g. by this person's CLI command) only a few AVR109 command letters may be entered without the risc of getting into trouble:

V, v, p

S (display bootloader name and set timeout to $>$ 1 minute) and most important in that situation

E (exit bootloader and start system / application software).

But do not enter (small) e inadvertently — except you want to erase flash except the bootloader itself and hence kill all else system and application software.

If the bootloader is to be build or used for other platforms, like say AurduinoMega2560 / ATmega2560 or for other com-ports make the applicable replacements, like e.g.

```
avrdude -p atmega2560  -c avr109 -b 38400   -P com6  -v -t
```

Defines

- #define getFlashByte(addr)

  Read a flash byte also from high addresses.
- #define hwCritEntBoot()

Hardware criterion for entering bootloader after reset.
- #define PARTCODE 0x44

  The (outdated) single byte part code.
- #define PROG_HW_VER

  The programmerHWver.
- #define PROG_SIGN "AVRBOOT"

  The programmerSign.
- #define PROG_SW_VER "10"

  The programmerSWver.
- #define SEC_with_F0 (72-61)

  timeout value for timOutCountPatr
- #define SEC_with_F8 (75-56)

  timeout value for timOutCountPatr
- #define SEC_with_FF (72-35)

  timeout value for timOutCountPatr
- #define TIMEOUT_LIGHTSHOW

  Time out HMI display.

Functions

- ADDR_T blockLoadEE (uint16_t size, ADDR_T address)

  Write a sequence of bytes read from UART(0) to EEPROM.
- ADDR_T blockLoadFl (uint16_t size, ADDR_T address)

  Write a sequence of words read from UART(0) to flash memory.
- ADDR_T blockReadEE (uint16_t size, ADDR_T address)

  Read a sequence of bytes from EEPROM and send them via UART(0)
- ADDR_T blockReadFl (uint16_t size, ADDR_T address)

  Read a sequence of bytes from flash memory and send them via UART(0)
- void bootLoaderGreet (void)

  Send the greeting lines for bootloader's start.
- uint8_t isFlashCleared (void)

  Flash memory has no program.
- void setTheLed (uint8_t state)

  Set the LED.

Variables

- char const bootAut [ ]

  The author of weAutSys and its bootloader.
- char bootBld [ ]

  The build date and time.
- char const bootCop [ ]

  The copyright notice for weAutSys and its bootloader.
- char const bootRevDat [ ]

  The bootloader's revision and date.
- char const programmerHWver [ ]

  The programmers hardware version.
- char const programmerSign [ ]

  Software identifier/programmer signature.
- char const programmerSWver [ ]

  The programmers software version.
- uint8_t const timOutCountPatr [ ]

  An eight bit count down display pattern.

2.36.2 Define Documentation

2.36.2.1 #define getFlashByte(*addr*)

Read a flash byte also from high addresses.

Depending on the µController having a large flash memory or not this macro just delegates to pgm_read_byte_far or pgm_read_byte_near() respectively.

See also

ADDR_T

2.36.2.2 #define hwCritEntBoot( )

Hardware criterion for entering bootloader after reset.

For weAut_01 this is a pressed enter key while leaving reset (i.e. releasing the reset key or applying supply).

2.36.3 Function Documentation

2.36.3.1 void setTheLed (uint8_t *state*)

Set the LED.

If no test LED is available on the respective platform, this function does nothing. If more than one LED is available (as with weAut01) the most prominent red one is used. With weAut01 this is the red status LED (setStatusLedRd).

Parameters

state	0 / false: turn off; != 0: turn on

2.36.3.2 void bootLoaderGreet (void)

Send the greeting lines for bootloader's start.

This function sends some lines of text over the bootloader's serial link. These texts (bootloaderWlc, bootloaderPlatf, bootRevDat etc.) are to introduce the serial bootloader.

See also

sendSerBytes_P
sendSerBytes

2.36.3.3 ADDR_T blockLoadFl (uint16_t *size*, ADDR_T *address*)

Write a sequence of words read from UART(0) to flash memory.

This is a bootloader support function. It will read size bytes from the UART and transfer them to the program memory starting at `address`. The address parameter must be a word address. It value incremented by `size/2` will be returned.

Parameters

size	the number of bytes to write to flash memory. It must be an even number
address	the start word address to write to in flash memory. (It is an even byte address.)

Returns

address + size/2. That is the (incremented) address behind the last flash word modified

2.36.3.4 ADDR_T blockLoadEE (uint16_t *size*, ADDR_T *address*)

Write a sequence of bytes read from UART(0) to EEPROM.

This is a bootloader support function. It will read `size` bytes from the UART and transfer them to the EEPROM memory starting at `address`. The address parameter is a byte address.

Parameters

size	the number of bytes to write to EEPROM
address	the start address (of the first EEPROM byte to be modified). The parameters's type will fit the flash size even if EEPROMs are much smaller.

Returns

address + size. That is the (incremented) address behind the last EEPROM byte modified

2.36.3.5 ADDR_T blockReadFl (uint16_t *size*, ADDR_T *address*)

Read a sequence of bytes from flash memory and send them via UART(0)

This is a bootloader support function. It will read size bytes from flash memory starting at `address` and send them as pure binary values over the UART. The address parameter must be a word address. It value incremented by `size/2` will be returned.

Parameters

size	the number of bytes to read from flash or program memory
address	the start word address to read from flash memory. (It is an even byte address.)

Returns

address + size/2. That is the (incremented) address behind the last flash word read

2.36.3.6 ADDR_T blockReadEE (uint16_t *size*, ADDR_T *address*)

Read a sequence of bytes from EEPROM and send them via UART(0)

This is a bootloader support function. It will read size bytes from EEPROM starting at `address` and send them over the UART.

Parameters

size	the number of bytes to write to EEPROM memory
address	pointer to the source memory's start address. It will be incremented.

Returns

address + size. That is the (incremented) address behind the last EEPROM byte read

2.36.3.7 uint8_t **isFlashCleared** (void)

Flash memory has no program.

This function returns true if the flash cannot have been programmed with an application software after an erase.

The criterion is simply flash[0] == 0xFFFF. It is, nevertheless, a good criterion as with this interrupt vector unprogrammed there will be no application.

2.36.4 Variable Documentation

2.36.4.1 char const **programmerSWver**[ ]

The programmers software version.

The version is always 2 characters, namely digits. The value is the same for all target platforms.

2.36.4.2 char const **programmerHWver**[ ]

The programmers hardware version.

The version is always 2 characters, namely digits. The value may depend on the target platform.

2.36.4.3 char const **programmerSign**[ ]

Software identifier/programmer signature.

The signature is always 7 characters. To be accepted by avrDude it has t start with AVR. The value is PROG_SIGN.

2.36.4.4 char const **bootRevDat**[ ]

The bootloader's revision and date.

The text starts with blank and ends in a linefeed.

2.36.4.5 char const **bootAut**[ ]

The author of weAutSys and its bootloader.

It is the author's name and his personal domain.

This is a string in (high) flash memory.

See also

copyChars_P()

2.36.4.6 char const **bootCop**[ ]

The copyright notice for weAutSys and its bootloader.

This is a string in (high) flash memory.

See also

copyChars_P()
getSomeCharsP(char∗, prog_char∗, uint8_t)

2.36.4.7 char bootBld[ ]

The build date and time.

It is the word "build" followed by the date and time where the C-preprocessor rolled over the (this) file boot109.h. This will be time where the bootloader was build.

This is a string in (high) flash memory.

See also

copyChars_P()

2.36.4.8 uint8_t const timOutCountPatr[ ]

An eight bit count down display pattern.

This is an array in (high!) flash memory. "High" means in the bootloader flash area. That will be above 64K for some µControllers and means having to use `pgm_read_byte_far(farAdd)` instead of just `pgm_read_byte(add16bit)` for access.

For every second counted down there is a 8 bit = 8 LED display pattern the last one being 0 / all OFF. The total length is 72 (for 72 seconds. For less than 72 seconds start at pattern &timOutCountPatr[72 -tim]. To start at a certain pattern use SEC_with_FF (all on), SEC_with_F8 (5 LEDs on) or SEC_with_F0 (4 LEDs on) for the timeout value `tim`.

See also

bootloaderWlc
FAR_ADD

2.37 Bootloader integration

2.37.1 Overview

The serial AVR109 bootloader is implemented as an integral part of weAutSys / weAut_01. It is designed to be (by fuse setting) the entry point for all reset or restart.

Hence the board should always come with this bootloader in its (upper flash) place and be fused to enter it on reset. All the basic (weAut_01) initialisations are then done anyway by the bootloader. On an external or power on reset this bootloader will after a quite short time without receiving ESC jump to the system/application software start, see Bootloader operation for details and Bootloader support for an overview and further references.

There the initialisation tasks already done may (and will) be omitted from the "normal" system software (according to the value of USE_BOOTLOADER). Additionally functions and constants provided by the integrated bootloader (see file boot109.h) may be linked to and used by system or application software. About building and linking the bootloader see the descriptive part of file boot109.c and download and read

Albrecht Weinert

`A serial bootloader for weAut_01, ArduinoMega and akin`

for more information on re-using bootloader functions in the application software.

Defines

- #define FAR_ADD(varHF)

    Generate a far address for high flash memory items.
- #define getSerByte()

    Basic function: UART(0) get one byte already received.
- #define isSerByteRec()

    Basic function: UART(0) has one byte received.
- #define LOW_ADD(varLF)

    Generate a far address for low flash memory items.

Functions

- void appMain (uint8_t init) __attribute__((noreturn))

    Jump to application program.
- void basicSystemInit (void)

    Initialise system resources.
- void bootMain (void) __attribute__((noreturn))

    Jump to the bootloader.
- char ∗ copyChars_P (char ∗dst, ADDR_T src, uint8_t mxLen)

    Copy a string from flash memory to RAM.
- void initUART0 (uint32_t baudRate, uint8_t x2, uint8_t parity, uint8_t stopBits, uint8_t useInt)

    Initialise the serial input (UART0)
- uint8_t recvErrorState (void)

    Get UART receive error status and flush receiver on error.
- uint8_t recvSerByte (void)

    Basic UART receive one byte.
- ADDR_T resetCauseText_P (uint8_t resetCauses)

    The reset cause text.
- void sendSerByte (uint8_t c)

    Basic UART send one byte (guarded)
- void sendSerBytes (char ∗src)

Basic UART0 send multiple bytes from RAM.
- void sendSerBytes_P (ADDR_T src)

  Basic UART send multiple bytes from flash.
- void toHMI8LEDchain (uint8_t val)

  Output to a chain of eight HMI/visible LEDs.
- void wait25 (void)

  A very basic delay function keeping the CPU busy for about 25µs.
- uint8_t waitSerByte (uint8_t tOut)

  Basic UART wait for a byte received.

Variables

- char const bootloaderPlatf []

  Bootloader's platform name and CPU frequency.
- char const bootloaderWlc []

  Bootloader's welcome greeting and copyright notice.
- char const bootResetCause0 []

  no reset cause: exit from from active bootloader or by command
- char const bootResetCause1 []

  reset cause: power on
- char const bootResetCause2 []

  reset cause: external
- char const bootResetCause4 []

  reset cause: brown out
- char const bootResetCause8 []

  reset cause: watchdog
- char const bootResetCauseG []

  reset cause: JTAG
- char const bootResetCauseN []

  reset cause: not known
- char const bootTextReset []

  The text "Reset by:".
- char const bootTextRevis []

  The text "Revision:".
- char const greetEmptFlash []

  Greeting for empty application flash.

2.37.2 Define Documentation

2.37.2.1 #define FAR_ADD(*varHF*)

Generate a far address for high flash memory items.

avr-gcc is restricted (may say dumb) enough always to deliver 16 bit addresses respectively pointers for variables even if they reside by INFLASH or PROGMEM in higher addresses. Note that the ATmega1284P e.g. has 128K flash memory and the ATmega2560 e.g. has 256Kbyte.

Hence if for example the boatloader flash array timOutCountPatr is finally at address 0x01e20c &timOutCountPatr[1] will (with avr-gcc) be 0x0e20d instead of 0x1e20d.

Using `FAR_ADD(timOutCountPatr[1])` supplies both the &-operator and the missing 0x10000 respectively 0x30000, correcting the error described.

For devices with less than 64KByte = 32KWords flash memory this macro's value is just `&`(varHF).

To get a 32 bit address for an item in the lowest 64kbyte of use the macro LOW_ADD instead.

Note that both macros FAR_ADD and LOW_ADD cover all flash of an ATmega1284P but only half of the A-Tmega2560's flash.

Note: Hopefully in future, avg-gcc will automatically switch to 24bit pointers on larger devices and automatically handle them correctly. What, in the end, we have higher level languages and their compilers for? But, alas, as of end 2015 this is still not the case.

Parameters

varHF	a variable in the highest 64Kbyte flash memory block. This covers any bootloader space. Do not supply a pointer; the & (i.e. the address operator) will be applied to the parameter.

2.37.2.2 #define LOW_ADD(*varLF*)

Generate a far address for low flash memory items.

This macro provides a 16 or 32 bit address (ADDR_T) for an item (a variable) in lowest 64Kbate segment of the memory. The upper 16 bits of the address provided this way are 0.

Use this macro to make a 16 or 32 bit address (ADDR_T) parameter for e.g. sendSerBytes_P. Giving a (16 bit) pointer here would work by the (correct) automatic cast, also — but with a compiler warning.

Parameters

varLF	a variable in the lowest 64Kbyte flash memory block. This covers the (start of) application space. Do not supply a pointer; the & (i.e. the address operator) will be applied to the parameter.

See also

FAR_ADD

Examples:

main.c.

2.37.2.3 #define isSerByteRec( )

Basic function: UART(0) has one byte received.

This expression is non-zero if the UART(0) has one received byte ready

2.37.2.4 #define getSerByte( )

Basic function: UART(0) get one byte already received.

This is essentially the unguarded version of recvSerByte(void). In other words it has to be guarded by waitSerByte(uint8_t tOut) or by isSerByteRec().

2.37.3 Function Documentation

2.37.3.1 void basicSystemInit (void)

Initialise system resources.

This function does the basic initialisations for the target hardware and runtime. It essentially disables all interrupts and initialises the ports and other resources except the UART (for that see initUART0).

The situations to use are

- after reset
- before a system re-initialisation

that is

- before entering the bootloader
- before leaving the bootloader (to enter application software)

2.37.3.2 void **toHMI8LEDchain** (uint8_t *val*)

Output to a chain of eight HMI/visible LEDs.

If the target hardware has a line of eight LEDs this function sets them according to parameter `val`. Otherwise nothing is done.

The pre-condition is these eight LEDs being independently controllable by software without any side-effects on I/O to an external process controlled by the target hardware.

With weAut_01 these are the eight green digital input (DI) display LEDs.

With ArduinoMega and easyAVR — assuming above conditions to be met — it might be just one eight bit output port named by the macro CHAIN8LED_PORT. If no setting is made for a chain of 8 independent LEDs and the macro NO_CHAIN8LED is defined an implementation may just do nothing (without compile time warning).

2.37.3.3 void **appMain** (uint8_t *init*)

Jump to application program.

First of all this function disables all interrupts and the watchdog and does a basicSystemInit() if `init` is not 0. Parameter `init` should only be OFF if a ref basicSystemInit "basic initialisation" has been done and those settings haven't been spoiled.

This function then initialises the UART (0) with or without interrupt usage depending on the platform.

With freshly set zeroReg and stackpointer this function will then go to the normal program's start in the flash memory. Hence it will never return to the caller.

Parameters

init	ON: do full basicSystemInit() also

See also

bootMain

2.37.3.4 void **bootMain** (void)

Jump to the bootloader.

At first this function disables all interrupts and the watchdog.

With freshly set zeroReg and stackpointer this function will then go to the bootloader's start in the flash memory. Hence this function will never return to the caller.

See also

appMain

2.37.3.5 void **initUART0** (uint32_t *baudRate,* uint8_t *x2,* uint8_t *parity,* uint8_t *stopBits,* uint8_t *useInt*)

Initialise the serial input (UART0)

UART 0 is initialised in asynchronous mode for receiving and transmitting (full duplex). A recommended baudRate is 38400; it's fast enough for communication and even boot loading and still robust enough for critical or long lines.

If in doubt x2 should be OFF. It is needed only for very high baudRates with respect to the CPU clock frequency (F_CPU). And some exotic baudrate F_CPU combinations get higher accuracy with x2 ON.

Parameters

baudRate	the rate desired (usually 38400)
x2	true: double speed (half receive samples)
parity	0=false: none; odd , even value: odd, even
stopBits	2: 2 stop bits, 1 [default]: 1 stop bit

See also

uartSetBaudDivide

Parameters

useInt	true: use UART interrupts

2.37.3.6 **ADDR_T resetCauseText_P** (uint8_t *resetCauses*)

The reset cause text.
According to the bits defined for the MCUSR register this function returns a (flash memory) pointer to

```
" power on    ", " external reset ", " power (brown) out ",
" watchdog reset    ", " jtag reset ", " no reset (by command?) "
 or " unknown cause ".
```

Parameters

resetCauses	The µC's reset info as by MCUSR

Returns

a 32 or 16 bit pointer (ADDR_T) to a 0-terminated String in high (bootloader) flash memory

2.37.3.7 void **wait25** (void)

A very basic delay function keeping the CPU busy for about 25µs.

Calling this function will keep the CPU busy for 500 cycles (two less for small flash devices) on a 20 MHz platform (weAut_01). That will take 25µs. On slower platforms with e.g. 16 MHz (Arduino) the number of cycles is, of course, proportionally reduced.

2.37.3.8 char∗ **copyChars_P** (char ∗ *dst,* **ADDR_T** *src,* uint8_t *mxLen*)

Copy a string from flash memory to RAM.

This function copies a String

- from flash memory pointed to by a 32 bit address `src`

- to a RAM buffer pointed to by (normal) pointer `dst`.

The function copies up to `mxLen-1` characters from `src` to `dst` and appends a trailing 0. Returned is the address of the last character in `dst` modified (i.e. the 0's address).

Rationale: Besides a variety of functions to handle and output strings in RAM weAutSys features a second (sub-) set of functions to handle / output strings in flash memory, their names usually ending in `P` or `_P`. Insofar these functions usually save the trouble of copying strings form flash to RAM (prior to output e.g.). But due to avr-gcc C pointer limitations the standard flash variable handling won't work above 64K byte, as is the case with bootloader variables on large flash devices. A third set of functions for those use cases can hardly be justified. Here the workaround is "copy to an intermediate RAM buffer and use RAM handling functions". That workaround is support by this function.

Parameters

dst	the destination to modify (in RAM, not null!)
src	the source (string) to copy from (in flash memory, not null!)
mxLen	the maximum number of characters to modify in `dst` including the trailing 0 appended. The maximum advance of the return value to parameter `dest`

Returns

dst + number of characters modified; NULL only if `dst` is NULL

See also

getSomeCharsP(char∗, prog_char∗, uint8_t)
FAR_ADD
LOW_ADD
resetCauseText_P()

2.37.3.9 void sendSerByte (uint8_t *c*)

Basic UART send one byte (guarded)

This is a very basic send function for the initialised UART(0). It will wait for the UART being ready to accept the next byte to be transmitted. If the UART is not initialised or handled correctly it may wait endlessly.

This functions uses spin waiting and hence the respective remarks for recvSerByte apply.

Parameters

c	the character or byte to be sent

2.37.3.10 void sendSerBytes_P (ADDR_T *src*)

Basic UART send multiple bytes from flash.

This is a very basic send function for the initialised UART. It uses spin waiting and hence the remarks for sendSer-Byte apply.

Parameters

src	pointer to the first character or byte to be sent in of the 0-terminated string. Must not be null.

See also

recvSerByte
FAR_ADD
LOW_ADD
sendSerBytes

Examples:

main.c.

2.37.3.11 void **sendSerBytes** (char ∗ *src*)

Basic UART0 send multiple bytes from RAM.

This is a very basic send function for the initialised UART. It uses spin waiting and hence is meant for situations without any threading support (or interrupt), as e.g. early initialisation phases or bootloaders.

Parameters

src	pointer to the first character or byte to be sent in of the 0-terminated string, must not be null

See also

recvSerByte
sendSerByte
sendSerBytes_P

2.37.3.12 uint8_t **waitSerByte** (uint8_t *tOut*)

Basic UART wait for a byte received.

This is a very basic receive function for the initialised UART. It uses spin waiting for a received byte timeout of tOut/10 seconds.

Cause of this spin waiting this function is meant for situations without any threading support (or interrupt), as e.g. early initialisation phases or bootloaders.

Parameters

tOut	a time out in 100 ms (1..254)

Returns

0: a byte was received; not 0: timeout

See also

recvSerByte

2.37.3.13 uint8_t **recvSerByte** (void)

Basic UART receive one byte.

This is a very basic receive function for the initialised UART. If no received byte is ready it will endlessly wait for it. To avoid this it may be guarded by waitSerByte — in which case the (unguarded) macro getSerByte() may also be used.

This function uses spin waiting and would hence be blocking all other activities. It is meant for situations without any threading support (or interrupts), as e.g. early initialisation phases or in bootloaders. And it is, of course, never to be used with an armed watchdog.

Returns

the character or byte received

See also

sendSerByte

2.37.3.14 uint8_t **recvErrorState** (void)

Get UART receive error status and flush receiver on error.

This function retrieves and returns the error status of an initialised UART(0). If no error occurred 0 is returned and nothing else done.

Otherwise the initialised UART's receiver is flushed.

Returns

0 on no error or at least one of bits 4 (framing error), 3 (overrun error) or 2 (parity error) is set

See also

recvSerByte

2.37.4 Variable Documentation

2.37.4.1 char const **bootloaderWlc**[ ]

Bootloader's welcome greeting and copyright notice.

This is a four line text in (high!) flash memory plus three starting and two trailing empty lines. It is meant for display as well as embedded copyright notice.

2.37.4.2 char const **greetEmptFlash**[ ]

Greeting for empty application flash.

This is a one line text in (high!) flash memory plus two starting and one trailing empty lines. It says:

remain in bootloader: application flash empty

2.37.4.3 char const **bootloaderPlatf**[ ]

Bootloader's platform name and CPU frequency.

This is a one line text in (high!) flash memory ending in a linefeed.

2.37.4.4 char const **bootTextRevis**[ ]

The text "Revision:".

This is a text block in (high!) flash memory (0-terminated).

2.37.4.5 char const bootTextReset[ ]

The text "Reset by:".

This is a text block in (high!) flash memory (0-terminated).

2.38 + + + weAutSys -- software incorporated from other sources + + +

2.38.1 Overview

weAutSys is the runtime for small network enabled automation modules; it has been developed by `weinert - automation` for its products.

Being a new development from scratch, weAutSys nevertheless incorporates some adapted / modified open source software if (and only if)

- it provides professional long term proven solutions for basic problems and
- its open source license is not infecting weAutSys' commercial licenses

Software incorporated this way is

- the parts used of avr gcc lib,
- Protothreads and
- one of Adam Dunkels' TCP/IP stacks
- ChaN's FAT implementation fatFS

Modules

- + + pt -- Adam Dunkels' lightweight Protothreads + +
- + + uIP -- Adam Dunkels' TCP/IP stack + +
- + + uIP -- Application (layer) support + +
- + + fatFS -- ChaN's FAT filesystem implementation + +

2.39 + + pt -- Adam Dunkels' lightweight Protothreads + +

2.39.1 Overview

weAutSys uses Protothreads as its multi-threading base for all automation / user cycles as well as for system, alarm and I/O handling threads. Adam Dunkels' invention is very lightweight in terms of RAM and processor time consumptions and thus best fit for µ-controller based automation systems.

In weAutSys Protothreads is used with only minimal modifications / additions to Adam Dunkels' ingenious original.

Note

Uppercase Protothreads means the ingenious basic software approach (by Adam Dunkel) while lower case protothread relates to a concrete (weAutSys') thread, thread data-structure or thread function.

In (mostly) Adam Dunkels' words (3 June 2006):

Protothreads are extremely lightweight stackless threads designed for severely memory constrained systems such as small embedded systems or sensor network nodes. Protothreads can be used with or without an underlying operating system.

Protothreads provides a blocking context on top of an event-driven system, without the overhead of per-thread stacks. The purpose of Protothreads is to implement sequential flow of control without complex state machines or full multi-threading.

Main features:

- No machine specific code — the protothreads library is pure C
- Does not use error-prone functions such as longjmp()
- Very small RAM overhead — only two bytes per protothread
- Provides blocking wait without full multi-threading or stack-switching
- Freely available under a BSD-like open source license

The Protothreads library is released under an open source BSD-style license that allows for both non-commercial and commercial usage. The only requirement is that credit is given.

The Protothreads library was written by Adam Dunkels `adam@sics.se` with support from Oliver Schmidt `ol.sc@web.de`.

Modules

- Local continuations

Files

- file pt.h

  Protothreads implementation.

Data Structures

- struct pt

  A protothread's (raw) data structure.

Defines

- #define PT_ENDED

  thread has finished
- #define PT_EXITED

  thread has finished
- #define PT_WAITING 0

  thread paused due to a condition
- #define PT_YIELDED

  Thread paused.
- #define ptfnct_t

  The return type of a protothread function.

Initialization

- #define PT_INIT(pt)

  Initialise a protothread.

Declaration and definition

- #define PT_BEGIN(pt)

  Declare the start of a protothread inside the protothread function.
- #define PT_END(pt)

  Declare the end of a protothread.

Blocked wait

- #define PT_WAIT_UNTIL(pt, condition)

  Block and wait until condition is true.
- #define PT_WAIT_WHILE(pt, cond)

  Block and wait while condition is true.

Hierarchical protothreads

- #define PT_WAIT_THREAD(pt, thread)

  Block and wait until a child protothread completes.
- #define PT_SPAWN(pt, child, thread)

  Spawn a child protothread and wait until it exits.

Exiting and restarting

- #define PT_RESTART(pt)

  Restart the protothread.
- #define PT_EXIT(pt)

  Exit the protothread.
- #define PT_LEAVE(pt)

  End the protothread.

Yielding from a protothread

- #define PT_YIELD(pt)

  Yield from the current protothread.
- #define PT_YIELD_UNTIL(pt, cond)

  Yield from the protothread until a condition occurs.
- #define PT_YIELD_WHILE(pt, condition)

  Yield while a condition is true.
- #define PT_WAIT_ASYIELD_WHILE(pt, condition)

  Wait while a condition is true mimicked as yield.
- #define PT_OR_YIELD_HERE(pt)

  Yield only if a previous conditional wait did not.
- #define PT_OR_YIELD_REENTER()

  Yield only if a previous conditional wait did not and re-enter that.

Typedefs

- typedef struct pt pt_t

  A protothread's (raw) data structure as type.

2.39.2 Define Documentation

2.39.2.1 #define ptfnct_t

The return type of a protothread function.

/note Within the PT_BEGIN PT_END braces a protothread function must not return directly (forgetting rare exceptions with lc-addrlabels). The return is done by Protothread operations.

Return values are PT_WAITING, PT_YIELDED, PT_EXITED and PT_ENDED.

The following holds:

a) PT_WAITING < PT_YIELDED < PT_EXITED < PT_ENDED

b) There is no real semantic difference between PT_EXITED and PT_ENDED

The knowledge a) is useful in evaluating return values.

The difference of PT_EXITED and PT_ENDED is almost insignificant:

The macro PT_END which returns PT_ENDED must appear exactly once at the end of a protothread function because it is the closing bracket to PT_BEGIN.

The macro PT_EXIT returning PT_EXITED may on the other hand appear as often as useful.

The effect of PT_EXIT and PT_END is exactly the same:

The thread does end meaning its next schedule will "land" / start thread work after PT_BEGIN. (Hence you will hardly find any code that won't OR PT_EXITED and PT_ENDED in concerning conditionals.

An extra macro PT_LEAVE returns PT_ENDED and can be used anywhere (saving jumps to PT_END just for the intended return value).

Examples:

main.c.

2.39.2.2 #define PT_YIELDED

Thread paused.

Yielding is mostly done as courtesy to other threads or to obey CPU usage trime limits. As Protothreads are non-preemptive this is the responsibility of every single thread function!

Insofar the semantic difference to waiting is the (extra) reason to pause the thread, lest it runs too long. In that sense yielding with an condition should yield at least once no matter the condition's state — as does the original Protothread.

On the other hand weAutSys needed a to convey the (semantic) difference on the reason for a wait on a condition:

- condition will fulfill automatically or
- the calling software must actively take action.

Herefore we use PT_YIELDED vs. PT_WAITING to transport this information via the (child) thread functions return value.

2.39.2.3 #define PT_EXITED

thread has finished

See also

ptfnct_t

Examples:

main.c.

2.39.2.4 #define PT_ENDED

thread has finished

See also

ptfnct_t

Examples:

main.c.

2.39.2.5 #define PT_INIT(pt)

Initialise a protothread.

Initialisation must be done prior to starting to execute the protothread.

Parameters

pt	A pointer to the protothread control structure.

See also

PT_SPAWN()

2.39.2.6 #define PT_BEGIN(pt)

Declare the start of a protothread inside the protothread function.

This macro is used to declare the starting point of a protothread. It should be placed at or near the start of the function in which the protothread runs respectively is implemented. PT_BEGIN() is the gridiron to the last set (left, yield, block) re-entry point.

All statements above the PT_BEGIN() invocation will be executed each time the protothread is scheduled.

The statements below PT_BEGIN() will be executed at the first entry into an initialised or restarted protothread.

Parameters

pt	the pointer to the protothread control structure.

Examples:

main.c.

2.39.2.7 #define PT_END(pt)

Declare the end of a protothread.

This macro is used for declaring that a protothread ends. It must always be used together with a matching PT_BEGIN() macro. A statement below won't be reached (except when labeled and gone to).

Parameters

pt	the pointer to the protothread control structure.

Examples:

main.c.

2.39.2.8 #define PT_WAIT_UNTIL(pt, *condition*)

Block and wait until condition is true.

This macro blocks the protothread until the specified condition is true.

Parameters

pt	pointer to the protothread control structure
condition	the condition (a boolean expression)

Examples:

main.c.

2.39.2.9 #define PT_WAIT_WHILE(pt, *cond*)

Block and wait while condition is true.

This function blocks and waits while condition is true. See PT_WAIT_UNTIL().

Parameters

pt	pointer to the protothread control structure
cond	the condition (a boolean expression)

2.39.2.10 #define PT_WAIT_THREAD(pt, *thread*)

Block and wait until a child protothread completes.

This macro schedules a child protothread. The current protothread will block until the child protothread completes.

Note

The child protothread must be manually initialised with the PT_INIT() function before this function is used.

Parameters

pt	A pointer to the (calling) protothread control structure.
thread	The child protothread with arguments

See also

PT_SPAWN()

2.39.2.11 #define PT_SPAWN(pt, *child*, *thread*)

Spawn a child protothread and wait until it exits.

This macro spawns a child protothread and waits until it exits. The macro can only be used within a protothread.

Parameters

pt	A pointer to the (calling) protothread control structure
child	A pointer to the child protothread's control structure
thread	The child protothread's function call with arguments

2.39.2.12 #define PT_RESTART(pt)

Restart the protothread.

This macro will block once and cause the running protothread to restart its execution at the place after PT_BEGIN().

Parameters

pt	A pointer to the protothread control structure

2.39.2.13 #define PT_EXIT(pt)

Exit the protothread.

This macro causes the protothread to exit. If the protothread was spawned by another protothread, the parent protothread will become unblocked and can continue to run.

Parameters

pt	A pointer to the (raw) protothread control structure

Examples:

main.c.

2.39.2.14 #define PT_LEAVE(pt)

End the protothread.

This macro causes the protothread to end normally. It has the same effect as PT_EXIT except returning PT_ENDED. Insofar it is equivalent to falling through to the end of the PT_BEGIN .. PT_END brace.

Parameters

pt	A pointer to the (raw) protothread control structure

2.39.2.15 #define PT_YIELD(pt)

Yield from the current protothread.

This function will yield the protothread, thereby allowing other processing to take place. As Protothreads are non-preemptive by nature unconditional yielding is one (and the only) way to dependably subdivide longer running tasks into junks of maximum allowed execution time.

Parameters

pt	A pointer to the protothread control structure

See also

PT_YIELD_UNTIL
PT_WAIT_ASYIELD_WHILE

Examples:

main.c.

2.39.2.16 #define PT_YIELD_UNTIL(pt, *cond*)

Yield from the protothread until a condition occurs.

This function will yield the protothread, until the specified condition evaluates to true.

The thread yields at least once, even if the condition `cond` is true at first arrival. This is a bit counter-intuitive but may be wanted in some circumstances; i.e if a cut / subdivision of the task's runtime is due anyway.

If no yield should take place if the condition is yet true (behaviour like `PT_WAIT_UNTIL` but with other returned state) use `PT_WAIT_ASYIELD_WHILE` with the inverse condition.

Parameters

pt	A pointer to the protothread control structure
cond	The condition.

See also

PT_YIELD

2.39.2.17 #define PT_YIELD_WHILE(pt, *condition*)

Yield while a condition is true.

This macro yields the protothread until and only if the specified condition is true.

- The difference to PT_WAIT_WHILE is the returned state.

  Semantically this is a wait returning PT_YIELDED.

- The difference to PT_YIELD_UNTIL is the more intuitive behaviour; i.e not yielding or just going ahead if `condition` is false from start.

Parameters

pt	pointer to the protothread control structure
condition	the condition (a boolean expression).

See also

PT_OR_YIELD_HERE

2.39.2.18 #define PT_WAIT_ASYIELD_WHILE(pt, *condition*)

Wait while a condition is true mimicked as yield.

This macro yields the protothread until and only if the specified condition is true.

- The difference to PT_WAIT_WHILE is the returned state.

  Semantically this is a wait returning PT_YIELDED.

- The difference to PT_YIELD_WHILE is the behaviour of not yielding or just going ahead if `condition` is false from start.

Parameters

pt	pointer to the protothread control structure
condition	the condition (a boolean expression).

See also

PT_OR_YIELD_HERE

Examples:

main.c.

2.39.2.19 #define PT_OR_YIELD_HERE(pt)

Yield only if a previous conditional wait did not.

This macro yields the protothread if and only if a previous conditional wait or wait_as_yield yield did not do so and just went ahead.

Parameters

pt	pointer to the protothread control structure

See also

PT_WAIT_ASYIELD_WHILE
PT_OR_YIELD_REENTER()

2.39.2.20 #define PT_OR_YIELD_REENTER( )

Yield only if a previous conditional wait did not and re-enter that.

This macro yields if and only if a previous conditional wait or wait_as_yield macro did not do so and just went ahead. In that case this yields the protothread.

Contrary to the else similar PT_OR_YIELD_HERE the resume point for the next schedule is **not** at this macro's (PT_OR_YIELD_REENTER) call but (back) at the previous block or yield. The rationale lies in those cases where that previous block or yield macro's conditions must be rechecked or its other (side) effects have to be repeated.

Hint: As this behavior is not intelligibly to the reader of the source code a clarifying comment is strongly recommended:

```
PT_YIELD_OUT_SPACE(&cliThread->pt, 31, cliThread->repStreams); // (A)
// if and only if (A) didn't yield, do yield here and now and
PT_OR_YIELD_REENTER(); // re-enter at (A) above on next schedule
```

See also

PT_WAIT_ASYIELD_WHILE
PT_OR_YIELD_HERE

Examples:

main.c.

2.40 Local continuations

2.40.1 Overview

Local continuations form the basis for implementing protothreads. A local continuation can be *set* in a specific function to capture the state of the function. After a local continuation has been set can be *resumed* in order to restore the state of the function at the point where the local continuation was set.

Files

- file lc-addrlabels.h

  Implementation of local continuations based on GCC's feature "Labels as values".
- file lc.h

  Local continuations.

2.41 + + uIP -- Adam Dunkels' TCP/IP stack + +

2.41.1 Overview

uIP is a platform independent TCP/IP stack implementation intended for small 8-bit and 16-bit micro-controllers. It provides the necessary protocols for Internet communication, with a very small code footprint and RAM requirements. It supports just one network interface controller (NIC) which fits weAut_01 and akin modules.

weAutSys adapts uIP to ATmega, AVR-C and to the ENC28J60 interface chip by appropriate driver and support functions. Besides the (macro) settings in the file uipopts.h some adjustments in the uIP code were indicated.

As of this writing the base for the weAutSys port is uIP V.1.0. With said adjustments and corrections it is, of course, re-published as open source.

Modules

- Architecture specific uIP functions and types
- uIP conversion functions
- uIP configuration functions
- uIP initialisation functions
- uIP device driver functions
- Variables for uIP device drivers
- uIP application functions
- Configuration options for uIP

Data Structures

- struct uip_eth_addr

  Representation of a 48-bit Ethernet address / MAC address.
- struct uip_icmpip_hdr

  The ICMP and IP headers.
- struct uip_tcpip_hdr

  The TCP and IP headers.
- struct uip_udpip_hdr

  The UDP and IP headers.

Defines

- #define UIP_APPDATA_SIZE

  The buffer size available for user data in the buffer uip_buf.
- #define UIP_CLOSED

  state value in uip_conn->tcpstateflags
- #define UIP_CLOSING

  state value in uip_conn->tcpstateflags
- #define UIP_DATA

  flag to uip_process(): incoming data in uip_buf
- #define UIP_ESTABLISHED

  state value in uip_conn->tcpstateflags
- #define UIP_FIN_WAIT_1

  state value in uip_conn->tcpstateflags
- #define UIP_FIN_WAIT_2

  state value in uip_conn->tcpstateflags

- #define UIP_LAST_ACK

    state value in uip_conn->tcpstateflags
- #define UIP_POLL_REQUEST

    flag to uip_process(): poll a connection
- #define UIP_STOPPED

    extra state bit in uip_conn->tcpstateflags
- #define UIP_SYN_RCVD

    state value in uip_conn->tcpstateflags
- #define UIP_SYN_SENT

    state value in uip_conn->tcpstateflags
- #define UIP_TIME_WAIT

    state value in uip_conn->tcpstateflags
- #define UIP_TIMER

    flag to uip_process(): periodic timer has fired
- #define UIP_TS_MASK

    mask for state values in uip_conn->tcpstateflags
- #define UIP_UDP_SEND_CONN

    flag to uip_process(): construct UDP datagram in uip_buf

Typedefs

- typedef uint16_t uip_ip4addr_t [2]

    Representation of an IP V4 address.
- typedef uint16_t uip_ip6addr_t [8]

    Representation of an IP (V6) address.
- typedef uip_ip4addr_t uip_ipaddr_t

    IP address is of type IP V4.

Functions

- uint16_t uip_chksum (uint16_t ∗buf, uint16_t len)

    Calculate the Internet checksum over a buffer.
- uint16_t uip_ipchksum (void)

    Calculate the IP header checksum of the packet header in uip_buf.
- void uip_process (uint8_t flag)

    The actual uIP function which does all the work.
- uint16_t uip_tcpchksum (void)

    Calculate the TCP checksum of the packet in uip_buf and uip_appdata.
- uint16_t uip_udpchksum (void)

    Calculate the UDP checksum of the packet in uip_buf and uip_appdata.
- void uipSecTick (void)

    One second tick for the uiP stack.

Variables

- struct uip_eth_addr actMACadd

    The (actual) MAC address.
- void ∗ uip_appdata

    Pointer to the application data in the packet buffer.

2.41.2 Define Documentation

2.41.2.1 #define UIP_APPDATA_SIZE

The buffer size available for user data in the buffer uip_buf.

This macro holds the available size for user data in uip_buf . It is intended to be used for checking bounds of available user data.

2.41.3 Typedef Documentation

2.41.3.1 typedef uint16_t uip_ip4addr_t[2]

Representation of an IP V4 address.

For IP V4 (used as of this writing in small modules like weAut_01) this is uIP's type uip_ipaddr_t being just an array uint16_t [2] in the end.

Note

An Internet address of e.g. 1.2.3.4 would reside in this array in the (wrong i.e. network i.e. big) endianess: { 0x0201 , 0x0403 } . Representation of an IP (V4) address

2.41.4 Function Documentation

2.41.4.1 void uip_process (uint8_t *flag*)

The actual uIP function which does all the work.

Parameters

flag	says on what reason this function is called

2.41.4.2 uint16_t uip_chksum (uint16_t * *buf,* uint16_t *len*)

Calculate the Internet checksum over a buffer.

The Internet checksum is the one's complement of the one's complement sum of all 16-bit words in the buffer.

See RFC1071.

See also the ENC28J60's data sheet. This Ethernet driver chip can calculate just this type of checksum as an extra service described under the the (miss-leading) header of "DMA functions". As a pity, uIP's internal proceedings calculate this checksum before the data will be put to the Ethernet chip respectively after the are copied out by the driver interface functions. So it would be a hard job with deep impact on uIP to have the chip calculate the IP checksum too en lieu of this function uip_chksum which is quite expensive on micro-controllers.

Parameters

buf	A pointer to the buffer over which the checksum is to be computed.
len	The length of the buffer over which the checksum is to be computed.

Returns

The Internet checksum of the buffer.

2.41.4.3 uint16_t uip_ipchksum (void)

Calculate the IP header checksum of the packet header in uip_buf.

The IP header checksum is the Internet checksum of the 20 bytes of the IP header.

Returns

The IP header checksum of the IP header in the uip_buf buffer.

2.41.4.4 uint16_t uip_tcpchksum (void)

Calculate the TCP checksum of the packet in uip_buf and uip_appdata.

The TCP checksum is the Internet checksum of data contents of the TCP segment, and a pseudo-header as defined in RFC793.

Returns

The TCP checksum of the TCP segment in uip_buf and pointed to by uip_appdata.

2.41.4.5 uint16_t uip_udpchksum (void)

Calculate the UDP checksum of the packet in uip_buf and uip_appdata.

The UDP checksum is the Internet checksum of data contents of the UDP segment, and a pseudo-header as defined in RFC768.

Note

The uip_appdata pointer that points to the packet data may point anywhere in memory, so it is not possible to simply calculate the Internet checksum of the contents of the uip_buf buffer.

Returns

The UDP checksum of the UDP segment in uip_buf and pointed to by uip_appdata.

2.41.4.6 void uipSecTick (void)

One second tick for the uiP stack.

This function has to be called by the runtime every second. (weAutSys does this in the 1s system thread.)

The implementation efficiently handles the ARP and DHCP timers.

2.41.5 Variable Documentation

2.41.5.1 void* uip_appdata

Pointer to the application data in the packet buffer.

This pointer points to the application data when the application is called. If the application wishes to send data, the application may use this space to write the data into before calling uip_send().

Examples:

main.c.

2.41.5.2 struct uip_eth_addr actMACadd

The (actual) MAC address.

It can be changed on initialisation and has to be held consistent with the ENC28J60's MAC setting.

This is the (one) MAC address used for the Ethernet link. It is to be known by the Ethernet stack uIP and by the driver (of ENC28J60).

2.42 Architecture specific uIP functions and types

2.42.1 Overview

The functions in this architecture specific module mainly implement the IP checksum and 32-bit additions. They have to bridge the gap between big endian network byte order and little endian used in weAut_01's ATmel and most other architectures as well as the differences between normal arithmetic and internet / RFC (check) sum definitions.

The IP checksum calculation is the most computationally expensive operation in the TCP/IP stack and it therefore pays off to implement this as efficiently as the underlying architecture reasonably permits. Hence this is one of the fields where weAutSys' implementation significantly uses (GCC inline) ASM.

The type definitions in this module map uIP types to appropriate types of the architecture, runtime system and tool chain.

Files

- file uip_mess.h

  weAutSys definition of system message texts for uIP

Functions

- uint16_t chksum (uint16_t sum, const uint8_t ∗data, uint16_t len)

  Calculate the Internet checksum over a buffer.

Variables

- char icmpNotIcmpEcho [ ]

  icmp: not icmp echo. (in flash)
- char icmpUnknownMess [ ]

  icmp: unknown ICMP message. (in flash IPV6)
- char ipBadCheckS [ ]

  ip: bad checksum. (in flash)
- char ipFragmDropp [ ]

  ip: fragment dropped. (in flash)
- char ipInvVersHeadL [ ]

  ip: invalid version or header length. (in flash)
- char ipNotTcpNorIcmp [ ]

  ip: neither tcp nor icmp. (in flash)
- char ipNotTcpNorIcmp6 [ ]

  ip: neither tcp nor icmp6. (in flash IPV6)
- char ipPackDroppSiAA [ ]

  ip: packet dropped since no address assigned. (in flash)
- char ipPackShorter [ ]

  ip: packet shorter than reported in IP header. (in flash)
- char ipPossPingConfR [ ]

  ip: possible ping config packet received. (in flash)
- char tcpBadCheckS [ ]

  tcp: bad checksum. (in flash)
- char tcpGotResetAbrt [ ]

  tcp: got reset, aborting connection. (in flash)
- char tcpNoUUsedCon [ ]

tcp: found no unused connections. (in flash)

- char udpBadCheckS []

  udp: bad checksum. (in flash)

- char udpNoMatchCon []

  udp: no connection matching (in flash)

2.42.2 Function Documentation

2.42.2.1 uint16_t **chksum** (uint16_t *sum*, const uint8_t ∗ *data*, uint16_t *len*)

Calculate the Internet checksum over a buffer.

This function adds the sum of all all 16-bit words in the buffer `data` to the parameter `sum` and returns the result.

`sum` and the returned result are in architecture byte order, being little endian (intel, ATmega, weAutSys — Automation runtime system weAut_01).

The 16 bit words in the buffer on the other hand are big endian. If `len` is odd the last byte in the buffer data is hence the high byte of word with an imaginary low byte 0x00.

Another peculiarity of this type of unsigned checksum is the feeding back of a carry out of any 16 bit addition step.

This characteristics are adversarial to higher programming languages even to C. For weAutSys this function is significantly improved by inline assembler.

Parameters

sum	the checksum's start value
data	a pointer to the buffer over which the checksum is to be computed
len	the length of the buffer

Returns

the checksum

2.43 uIP conversion functions

2.43.1 Overview

These functions can be used for converting between different data formats used by uIP.

Defines

- #define HTONS(n)

  Convert 16-bit quantity from host byte order to network byte order.
- #define uip_ip6addr(addr, addr0, addr1, addr2, addr3, addr4, addr5, addr6, addr7)

  Construct an IPv6 address from eight 16-bit words.
- #define uip_ipaddr(addr, addr0, addr1, addr2, addr3)

  Construct an IP address from four bytes.
- #define uip_ipaddr1(addr)

  Pick the first octet of an IP address.
- #define uip_ipaddr2(addr)

  Pick the second octet of an IP address.
- #define uip_ipaddr3(addr)

  Pick the third octet of an IP address.
- #define uip_ipaddr4(addr)

  Pick the fourth octet of an IP address.
- #define uip_ipaddr_cmp(addr1, addr2)

  Compare two IP addresses.
- #define uip_ipaddr_copy(dest, src)

  Copy an IP address to another IP address.
- #define uip_ipaddr_mask(dest, src, mask)

  Mask out the network part of an IP address.
- #define uip_ipaddr_maskcmp(addr1, addr2, mask)

  Compare two IP addresses with netmasks.

2.43.2 Define Documentation

2.43.2.1 #define uip_ipaddr(*addr, addr0, addr1, addr2, addr3*)

Construct an IP address from four bytes.

This function constructs an IP address of the type that uIP handles internally from four bytes. The function is handy for specifying IP addresses to use with e.g. the uip_connect() function.

Example:

```
uip_ipaddr_t ipaddr;
struct uip_conn *c;

uip_ipaddr(&ipaddr, 192,168,1,2);
c = uip_connect(&ipaddr, HTONS(80));
```

Parameters

addr	A pointer to a uip_ipaddr_t variable that will be filled in with the IP address.
addr0	The first octet of the IP address.
addr1	The second octet of the IP address.
addr2	The third octet of the IP address.
addr3	The forth octet of the IP address.

2.43.2.2 #define **uip_ip6addr(** *addr, addr0, addr1, addr2, addr3, addr4, addr5, addr6, addr7*)

Construct an IPv6 address from eight 16-bit words.

This function constructs an IPv6 address.

2.43.2.3 #define **uip_ipaddr_copy(** *dest, src*)

Copy an IP address to another IP address.

This is just a copy of 4 Bytes in the (default) case of IP V4 and 8 bytes in the case of IP V6 from `src` to `dest`.

No endianess is changed. The content of a type uip_ipaddr_t variable is considered to be in network byte order. Example:

```
uip_ipaddr_t ipaddr1, ipaddr2;

uip_ipaddr(&ipaddr1, 192,16,1,2);
uip_ipaddr_copy(&ipaddr2, &ipaddr1);
```

Parameters

dest	pointer to the destination to copy to
src	pointer to the source where to copy from

2.43.2.4 #define **uip_ipaddr_cmp(** *addr1, addr2*)

Compare two IP addresses.

Parameters

addr1	The first IP address.
addr2	The second IP address.

2.43.2.5 #define **uip_ipaddr_maskcmp(** *addr1, addr2, mask*)

Compare two IP addresses with netmasks.

The masks are used to mask out the bits that are to be compared.

Parameters

addr1	The first IP address.
addr2	The second IP address.
mask	The netmask.

2.43.2.6 #define **uip_ipaddr_mask(** *dest, src, mask*)

Mask out the network part of an IP address.
Example:

```
uip_ipaddr_t ipaddr1, ipaddr2, netmask;

uip_ipaddr(&ipaddr1, 192,16,1,2);
uip_ipaddr(&netmask, 255,255,255,0);
uip_ipaddr_mask(&ipaddr2, &ipaddr1, &netmask);
```

In the example above, the variable "ipaddr2" will contain the IP address 192.168.1.0.

Parameters

dest	Where the result is to be placed.
src	The IP address.
mask	The netmask.

2.43.2.7 #define uip_ipaddr1(*addr*)

Pick the first octet of an IP address.

Example:

```
uip_ipaddr_t ipaddr;
uint8_t octet;

uip_ipaddr(&ipaddr, 1,2,3,4);
octet = uip_ipaddr1(&ipaddr);
```

In the example above, the variable "octet" will contain the value 1.

2.43.2.8 #define uip_ipaddr2(*addr*)

Pick the second octet of an IP address.

Example:

```
uip_ipaddr_t ipaddr;
uint8_t octet;

uip_ipaddr(&ipaddr, 1,2,3,4);
octet = uip_ipaddr2(&ipaddr);
```

In the example above, the variable "octet" will contain the value 2.

2.43.2.9 #define uip_ipaddr3(*addr*)

Pick the third octet of an IP address.

Example:

```
uip_ipaddr_t ipaddr;
uint8_t octet;

uip_ipaddr(&ipaddr, 1,2,3,4);
octet = uip_ipaddr3(&ipaddr);
```

In the example above, the variable "octet" will contain the value 3.

2.43.2.10 #define uip_ipaddr4(*addr*)

Pick the fourth octet of an IP address.

Example:

```
uip_ipaddr_t ipaddr;
uint8_t octet;

uip_ipaddr(&ipaddr, 1,2,3,4);
octet = uip_ipaddr4(&ipaddr);
```

In the example above, the variable "octet" will contain the value 4.

2.43.2.11 #define HTONS(*n*)

Convert 16-bit quantity from host byte order to network byte order.

This macro is primarily used for converting constants from host byte order to network byte order. For converting variables to network byte order, use the htons() function instead.

Examples:

main.c.

2.44 uIP configuration functions

2.44.1 Overview

The uIP configuration functions are used for setting run-time parameters in uIP such as IP addresses.

Defines

- #define uip_getdraddr(addr)

  Get the default router's IP address.
- #define uip_gethostaddr(addr)

  Get the IP address of this host.
- #define uip_getnetmask(addr)

  Get the netmask.
- #define uip_setdraddr(addr)

  Set the default router's IP address.
- #define uip_sethostaddr(addr)

  Set the IP address of this host.
- #define uip_setnetmask(addr)

  Set the netmask.

2.44.2 Define Documentation

2.44.2.1 #define uip_sethostaddr(*addr*)

Set the IP address of this host.

The IP address is represented as a 4-byte array where the first octet of the IP address is put in the first member of the 4-byte array.

Example:

```
uip_ipaddr_t addr;

uip_ipaddr(&addr, 192,168,1,2);
uip_sethostaddr(&addr);
```

Parameters

addr	A pointer to an IP address of type uip_ipaddr_t

See also: uip_ipaddr()

2.44.2.2 #define uip_gethostaddr(*addr*)

Get the IP address of this host.

The IP address is represented as a 4-byte array where the first octet of the IP address is put in the first member of the 4-byte array.

Example:

```
uip_ipaddr_t hostaddr;

uip_gethostaddr(&hostaddr);
```

Parameters

addr	A pointer to a uip_ipaddr_t variable that will be filled in with the currently configured IP address.

2.44.2.3 #define uip_setdraddr(*addr*)

Set the default router's IP address.

Parameters

addr	A pointer to a uip_ipaddr_t variable containing the IP address of the default router.

See also

uip_ipaddr()

2.44.2.4 #define uip_setnetmask(*addr*)

Set the netmask.

Parameters

addr	A pointer to a uip_ipaddr_t variable containing the IP address of the netmask.

See also: uip_ipaddr()

2.44.2.5 #define uip_getdraddr(*addr*)

Get the default router's IP address.

Parameters

addr	A pointer to a uip_ipaddr_t variable that will be filled in with the IP address of the default router.

2.44.2.6 #define uip_getnetmask(*addr*)

Get the netmask.

Parameters

addr	A pointer to a uip_ipaddr_t variable that will be filled in with the value of the netmask.

2.45 uIP initialisation functions

2.45.1 Overview

The uIP initialisation functions are used for booting uIP.

Functions

- void uip_init (void)

  uIP initialisation function
- void uip_setipid (uint16_t id)

  uIP initialisation function

2.45.2 Function Documentation

2.45.2.1 void **uip_init** (void)

uIP initialisation function

This function should be called at boot up to initialize the uIP TCP/IP stack.

2.45.2.2 void **uip_setipid** (uint16_t *id*)

uIP initialisation function

This function may be used at boot time to set the initial ip_id which is an increasing number that is used for the IP ID field.

2.46 uIP device driver functions

2.46.1 Overview

These functions are used by a network device driver for interacting with uIP.

Defines

- #define uip_conn_active(conn)

  Check if a connection identified by its number is active.
- #define uip_input()

  Process an incoming packet.
- #define uip_periodic(conn)

  Periodic processing for a connection identified by its number.
- #define uip_periodic_conn(conn)

  Perform periodic processing for a connection identified by pointer.
- #define uip_poll_conn(conn)

  Request that a particular connection should be polled.

Variables

- uint8_t uip_buf [ ]

  The uIP packet buffer.
- char ∗ uip_buf_alias

  alias to uip_buf

2.46.2 Define Documentation

2.46.2.1 #define uip_input( )

Process an incoming packet.

This function should be called when the device driver has received a packet from the network. The packet from the device driver must be present in the uip_buf buffer, and the length of the packet should be placed in the uip_len variable.

When the function returns, there may be an outbound packet placed in the uip_buf packet buffer. If so, the uip_len variable is set to the length of the packet. If no packet is to be sent out, the uip_len variable is set to 0.

The usual way of calling the function is presented by the source code below.

```
uip_len = devicedriver_poll();
if(uip_len > 0) {
  uip_input();
  if(uip_len > 0) {
    devicedriver_send();
  }
}
```

Note

If you are writing a uIP device driver that needs ARP (Address Resolution Protocol), e.g., when running uIP over Ethernet, you will need to call the uIP ARP code before calling this function:

```
#define BUF ((struct uip_eth_hdr *)&uip_buf[0])
uip_len = ethernet_devicedrver_poll();
if(uip_len > 0) {
  if(BUF->type == HTONS(UIP_ETHTYPE_IP)) {
    uip_arp_ipin();
    uip_input();
    if(uip_len > 0) {
      uip_arp_out();
 ethernet_devicedriver_send();
    }
  } else if(BUF->type == HTONS(UIP_ETHTYPE_ARP)) {
    uip_arp_arpin();
    if(uip_len > 0) {
 ethernet_devicedriver_send();
    }
  }
```

2.46.2.2 #define uip_periodic(*conn*)

Periodic processing for a connection identified by its number.

This function does the necessary periodic processing (timers, polling) for a uIP TCP connection, and should be called when the periodic uIP timer goes off. It should be called for every connection, regardless of whether they are open or closed.

When the function returns, it may have an outbound packet waiting for service in the uIP packet buffer, and if so the uip_len variable is set to a value larger than zero. The device driver should hence be called to send out the packet.

The usual way of calling the function is through a for() loop like this:

```
for(i = 0; i < UIP_CONNS; ++i) {
  uip_periodic(i);
  if(uip_len > 0) {
    devicedriver_send();
  }
}
```

Note

If you are writing a uIP device driver that needs ARP (Address Resolution Protocol), e.g., when running uIP over Ethernet, you will need to call the uip_arp_out() function before calling the device driver:

```
for(i = 0; i < UIP_CONNS; ++i) {
  uip_periodic(i);
  if(uip_len > 0) {
    uip_arp_out();
    ethernet_devicedriver_send();
  }
}
```

Parameters

conn	number of the connection which is to be periodically polled

See also

uip_conn_active

2.46.2.3 #define uip_periodic_conn(*conn*)

Perform periodic processing for a connection identified by pointer.

Same as uip_periodic() but takes a pointer to the actual uip_conn struct instead of its number. This function can be used to force periodic processing of a specific connection.

See also: uip_periodic

Parameters

conn	A pointer to the uip_conn struct of the connection

2.46.2.4 #define uip_poll_conn(*conn*)

Request that a particular connection should be polled.

Similar to uip_periodic_conn() but does not perform any timer processing. The application is polled for new data.

Parameters

conn	A pointer to the uip_conn struct for the connection to be processed.

2.46.3 Variable Documentation

2.46.3.1 uint8_t uip_buf[ ]

The uIP packet buffer.

The uip_buf array is used to hold incoming and outgoing packets. The device driver should place incoming data into this buffer. When sending data, the device driver should read the link level headers and the TCP/IP headers from this buffer. The size of the link level headers is configured by the UIP_LLH_LEN define.

Note

The application data need not be placed in this buffer, so the device driver must read it from the place pointed to by the uip_appdata pointer as illustrated by the following example:

```
void
devicedriver_send(void)
{
   hwsend(&uip_buf[0], UIP_LLH_LEN);
   if(uip_len <= UIP_LLH_LEN + UIP_TCPIP_HLEN) {
     hwsend(&uip_buf[UIP_LLH_LEN], uip_len - UIP_LLH_LEN);
   } else {
     hwsend(&uip_buf[UIP_LLH_LEN], UIP_TCPIP_HLEN);
     hwsend(uip_appdata, uip_len - UIP_TCPIP_HLEN - UIP_LLH_LEN);
   }
}
```

2.47 Variables for uIP device drivers

2.47.1 Overview

uIP has a few global variables that are used in device drivers for uIP.

Data Structures

- struct uipConn_t

  Representation of a uIP TCP connection.

Variables

- struct uipConn_t ∗ uip_conn

  Pointer to the current TCP connection.
- struct uipConn_t uip_conns [ ]

  The array containing all uIP connections.
- uint16_t uip_len

  The length of the packet in the uip_buf buffer.

2.47.2 Variable Documentation

2.47.2.1 uint16_t uip_len

The length of the packet in the uip_buf buffer.

The global variable uip_len holds the length of the packet in the uip_buf buffer.

When the network device driver calls the uIP input function, uip_len should be set to the length of the packet in the uip_buf buffer.

When sending packets, the device driver should use the contents of the uip_len variable to determine the length of the outgoing packet.

2.47.2.2 struct uipConn_t∗ uip_conn

Pointer to the current TCP connection.

The uip_conn pointer can be used to access the current TCP connection

Examples:

main.c.

2.47.2.3 struct uipConn_t uip_conns[ ]

The array containing all uIP connections.

See also: UIP_CONNS

2.48 uIP application functions

2.48.1 Overview

Functions used by an application running of top of uIP.

These functions do the opening and closing of connections, sending and receiving data, etc.

Defines

- #define uip_abort()

  Abort the current connection.
- #define uip_aborted()

  Has the connection been aborted by the other end.
- #define uip_acked()

  Has previously sent data been acknowledged?
- #define uip_close()

  Close the current connection.
- #define uip_closed()

  Has the connection been closed by the other end?
- #define uip_connected()

  Has the connection just been connected?
- #define uip_datalen()

  The length of any incoming data that is currently available (if available) in the uip_appdata buffer.
- #define uip_initialmss()

  Get the initial maximum segment size (MSS) of the current connection.
- #define uip_mss()

  Current maximum segment size that can be sent on the current connection.
- #define uip_newdata()

  Is new incoming data available?
- #define uip_poll()

  Is the connection being polled by uIP?
- #define uip_restart()

  Restart the current connection, if is has previously been stopped with uip_stop()
- #define uip_rexmit()

  Do we need to retransmit previously data?
- #define uip_stop()

  Tell the sending host to stop sending data.
- #define uip_stopped(conn)

  Find out if the current connection has been previously stopped with uip_stop()
- #define uip_timedout()

  Has the connection timed out?
- #define uip_udp_bind(conn, port)

  Bind a UDP connection to a local port.
- #define uip_udp_remove(conn)

  Remove / give up an UDP connection.
- #define uip_udp_send(len)

  Send a UDP datagram of length len on the current connection.
- #define uip_udpconnection()

  Is the current connection a UDP connection?

Functions

- struct uipConn_t ∗ uip_connect (uip_ipaddr_t ∗ripaddr, uint16_t port)

  Connect to a remote host using TCP.
- void uip_listen (uint16_t port)

  Start listening on the specified port.
- void uip_send (const void ∗data, int len)

  Send data on the current connection.
- struct uip_udp_conn ∗ uip_udp_new (uip_ipaddr_t ∗ripaddr, uint16_t rport)

  Set up a new UDP connection.
- void uip_unlisten (uint16_t port)

  Stop listening on the specified port.

2.48.2 Define Documentation

2.48.2.1 #define uip_datalen( )

The length of any incoming data that is currently available (if available) in the uip_appdata buffer.

This returns the length of any incoming data that is currently available in the uip_appdata buffer.

If works only if there is any data available at all: the test function uip_newdata() must first be used to check this condition.

Examples:

main.c.

2.48.2.2 #define uip_close( )

Close the current connection.

This function will close the current connection in a nice way.

2.48.2.3 #define uip_abort( )

Abort the current connection.

This function will abort (reset) the current connection, and is usually used when an error has occurred that prevents using the uip_close() function.

2.48.2.4 #define uip_stop( )

Tell the sending host to stop sending data.

This function will close our receiver's window so that we stop receiving data for the current connection.

2.48.2.5 #define uip_restart( )

Restart the current connection, if is has previously been stopped with uip_stop()

This function will open the receiver's window again so that we start receiving data for the current connection.

2.48.2.6 #define uip_udpconnection( )

Is the current connection a UDP connection?

This function checks whether the current connection is a UDP connection.

2.48.2.7 #define uip_newdata( )

Is new incoming data available?

Will return non-zero resp. true if there is new data for the application present at the uip_appdata pointer. The size of the data is available through the uip_len variable.

Examples:

main.c.

2.48.2.8 #define uip_acked( )

Has previously sent data been acknowledged?

Will return non-zero if the previously sent data has been acknowledged by the remote host. This means that the application can send new data.

Examples:

main.c.

2.48.2.9 #define uip_connected( )

Has the connection just been connected?

Will return non-zero if the current connection has been connected to a remote host. This will happen both if the connection has been actively opened (with uip_connect()) or passively opened (with uip_listen()).

2.48.2.10 #define uip_closed( )

Has the connection been closed by the other end?

Is non-zero if the connection has been closed by the remote host. The application may then do the necessary clean-ups.

2.48.2.11 #define uip_aborted( )

Has the connection been aborted by the other end.

Non-zero if the current connection has been aborted (reset) by the remote host.

2.48.2.12 #define uip_timedout( )

Has the connection timed out?

Non-zero if the current connection has been aborted due to too many retransmissions.

2.48.2.13 #define uip_rexmit( )

Do we need to retransmit previously data?

Returns non-zero if the previously sent data has been lost in the network, and the application should retransmit it. The application should send exactly the same data as it did the last time, using the uip_send() function.

Examples:

main.c.

2.48.2.14 #define uip_poll( )

Is the connection being polled by uIP?

Is non-zero if the reason the application is invoked is that the current connection has been idle for a while and should be polled.

The polling event can be used for sending data without having to wait for the remote host to send data.

2.48.2.15 #define uip_mss( )

Current maximum segment size that can be sent on the current connection.

The current maximum segment size that can be sent on the connection is computed from the receiver's window and the MSS of the connection (which also is available by calling uip_initialmss()).

2.48.2.16 #define uip_udp_remove(*conn*)

Remove / give up an UDP connection.

Parameters

conn	pointer to the uip_udp_conn structure for the connection

2.48.2.17 #define uip_udp_bind(*conn,* *port*)

Bind a UDP connection to a local port.

Parameters

conn	pointer to the uip_udp_conn structure for the connection
port	The local port number, in network byte order

2.48.2.18 #define uip_udp_send(*len*)

Send a UDP datagram of length len on the current connection.

This function can only be called in response to a UDP event (poll or newdata). The data must be present in the uip_buf buffer, at the place pointed to by the uip_appdata pointer.

Parameters

len	The length of the data in the uip_buf buffer

2.48.3 Function Documentation

2.48.3.1 void **uip_listen** (uint16_t *port*)

Start listening on the specified port.

Note

Since this function expects the port number in network byte order, a conversion using HTONS() or htons() is necessary.

```
uip_listen(HTONS(80));
```

Parameters

port	A 16-bit port number in network byte order.

Examples:

main.c.

2.48.3.2 void **uip_unlisten** (uint16_t *port*)

Stop listening on the specified port.

Note

Since this function expects the port number in network byte order, a conversion using HTONS() or htons() is necessary.

```
uip_unlisten(HTONS(80));
```

Parameters

port	A 16-bit port number in network byte order.

2.48.3.3 struct **uipConn_t**∗ **uip_connect** (**uip_ipaddr_t** ∗ *ripaddr*, uint16_t *port*) [read]

Connect to a remote host using TCP.

This function is used to start a new connection to the specified port on the specified host. It allocates a new connection identifier, sets the connection to the SYN_SENT state and sets the retransmission timer to 0. This will cause a TCP SYN segment to be sent out the next time this connection is periodically processed, which usually is done within 0.5 seconds after the call to uip_connect().

Note

This function is available only if support for active open has been configured by defining UIP_ACTIVE_OPEN to 1 in uipopt.h.
Since this function requires the port number to be in network byte order, a conversion using HTONS() or htons() is necessary.

```
uip_ipaddr_t ipaddr;

uip_ipaddr(&ipaddr, 192,168,1,2);
uip_connect(&ipaddr, HTONS(80));
```

Parameters

ripaddr	the IP address of the remote hot.
port	a 16-bit port number in network byte order.

Returns

a pointer to the uIP connection identifier for the new connection, or NULL if no connection could be allocated.

2.48.3.4 void uip_send (const void ∗ *data,* int *len*)

Send data on the current connection.

This function is used to send out a single segment of TCP data. Only applications that have been invoked by uIP for event processing can send data.

The amount of data that actually is sent out after a call to this function is determined by the maximum amount of data TCP allows. uIP will automatically crop the data so that only the appropriate amount of data is sent. The function uip_mss() can be used to query uIP for the amount of data that actually will be sent.

Note

This function does not guarantee that the sent data will arrive at the destination. If the data is lost in the network, the application will be invoked with the uip_rexmit() event being set. The application will then have to re-send the data using this function.

Parameters

data	A pointer to the data which is to be sent Using the buffer pointed to by uip_appdata to prepare the send data (and then use as this parameter) is strongly recommended.
len	The maximum amount of data bytes to be sent. Due to internal buffer limits this must not be more than UIP_APPDATA_SIZE

Examples:

main.c.

2.48.3.5 struct uip_udp_conn∗ uip_udp_new (uip_ipaddr_t ∗ *ripaddr,* uint16_t *rport*) [read]

Set up a new UDP connection.

This function sets up a new UDP connection. The function will automatically allocate an unused local port for the new connection. However, another port can be chosen by using the uip_udp_bind() call, after the uip_udp_new() function has been called.

Example:

```
uip_ipaddr_t addr;
struct uip_udp_conn *c;

uip_ipaddr(&addr, 192,168,2,1);
c = uip_udp_new(&addr, HTONS(12345));
if(c != NULL) {
  uip_udp_bind(c, HTONS(12344));
}
```

Parameters

ripaddr	the IP address of the remote host
rport	the remote port number in network byte order

Returns

the uip_udp_conn structure for the new connection or NULL if no connection could be allocated.

2.49 + + uIP -- Application (layer) support + +

2.49.1 Overview

uIP is basically Adam Dunkels' TCP/IP small footprint stack.

At the upper stack levels uIP brings some protocol and application support. weAutSys has adapted and uses ARP, DNS (resolve) and the DHCP client and brings further applications like NTP, Telnet and Modbus.

Modules

- uIP Address Resolution Protocol
- DHCP client Dynamic Host Configuration Protocol
- DNS resolver

2.50 uIP Address Resolution Protocol

2.50.1 Overview

The Address Resolution Protocol ARP is used for mapping between IP addresses and link level addresses such as the Ethernet MAC addresses. ARP uses broadcast queries to ask for the link level address of a known IP address and the host which is configured with the IP address for which the query was meant, will respond with its link level address.

Note

uIP's ARP implementation only supports Ethernet.

The ARP code needs to know the MAC address of the Ethernet card in order to be able to respond to ARP queries and to generate working Ethernet headers. So actMACadd has to be set to the correct value before the ARP implementation can work.

Files

- file uip_arp.h

  Definitions for the ARP module.

Data Structures

- struct arp_entry

  An ARP entry.
- struct arp_hdr

  The ARP header.
- struct ethip_hdr

  The IP header.
- struct uip_eth_hdr

  The Ethernet header.

Defines

- #define ARP_HWTYPE_ETH

  ARP packet hardware Ethernet (the only supported one)
- #define ARP_REPLY

  packet type answer "I have IP.. and MAC.."
- #define ARP_REQUEST

  packet type request "which MAC has IP...?"
- #define UIP_ETHTYPE_ARP

  Type ARP (in little endian)
- #define UIP_ETHTYPE_IP

  Type IP (V.4) (in little endian)
- #define UIP_ETHTYPE_IP6

  Type IPV6 (in little endian)

Functions

- void uip_arp_arpin (void)
  ARP processing for incoming ARP packets.
- void uip_arp_init (void)
  Initialise the ARP module.
- void uip_arp_ipin (void)
  ARP processing for incoming IP packets.
- void uip_arp_out (void)
  The ARP output prepare function.
- void uip_arp_timer (void)
  Periodic ARP processing function.

Variables

- struct arp_entry arp_table [ ]
  The ARP table.
- struct uip_eth_addr broadcast_ethaddr
  The broadcast MAC.
- const uint16_t broadcast_ipaddr [ ]
  The broadcast IP address.

2.50.2 Function Documentation

2.50.2.1 void **uip_arp_init** (void)

Initialise the ARP module.

This functions just clears all entries in the ARP table and the ARP time tick.

It should be called before calling any other ARP functions.

2.50.2.2 void **uip_arp_ipin** (void)

ARP processing for incoming IP packets.

This function should be called by the device driver when an IP packet has been received. The function will check if the address is in the ARP cache, and if so the ARP cache entry will be refreshed. If no ARP cache entry was found, a new one is created.

This function expects an IP packet with a prepended Ethernet header in the uip_buf[] buffer, and the length of the packet in the global variable uip_len.

2.50.2.3 void **uip_arp_arpin** (void)

ARP processing for incoming ARP packets.

This function should be called by the device driver when an ARP packet has been received. The function will act differently depending on the ARP packet type: if it is a reply for a request that we previously sent out, the ARP cache will be filled in with the values from the ARP reply. If the incoming ARP packet is an ARP request for our IP address, an ARP reply packet is created and put into the uip_buf[] buffer.

When the function returns, the value of the global variable uip_len indicates whether the device driver should send out a packet. If uip_len is zero, no packet should be sent. If uip_len is non-zero, it contains the length of the outbound packet that is present in the uip_buf[] buffer.

This function expects an ARP packet with a prepended Ethernet header in the uip_buf[] buffer, and the length of the packet in the global variable uip_len.

2.50.2.4 void **uip_arp_out** (void)

The ARP output prepare function.

It should be called before sending out an IP packet. The function checks the destination IP address of the IP packet to see what Ethernet MAC address that should be used as a destination MAC address on the Ethernet.

If the destination IP address is in the local network (determined by logical ANDing of netmask and our IP address), the function checks the ARP cache to see if an entry for the destination IP address is found. If so, an Ethernet header is prepended and the function returns. If no ARP cache entry is found for the destination IP address, the packet in the uip_buf[] is replaced by an ARP request packet for the IP address. The IP packet is dropped and it is assumed that the higher level protocols (e.g., TCP) eventually will retransmit the dropped packet.

If the destination IP address is not on the local network, the IP address of the default router is used instead.

When the function returns, a packet is present in the uip_buf[] buffer, and the length of the packet is in the global variable uip_len.

2.50.2.5 void **uip_arp_timer** (void)

Periodic ARP processing function.

This function performs periodic timer processing in the ARP module and should be called at regular intervals. It will clear old (out-dated) entries in the ARP table.

The recommended frequency is every 10 seconds.

See also: UIP_ARP_MAXAGE

2.50.3 Variable Documentation

2.50.3.1 struct **arp_entry arp_table**[ ]

The ARP table.

See also: UIP_ARPTAB_SIZE

2.51 DHCP client Dynamic Host Configuration Protocol

2.51.1 Overview

The Dynamic Host Configuration Protocol (DHCP) is used for getting IP configuration data from a server.

This is Adam Dunkels' DHCP client implementation on top of his uIP TCP/IP stack — Albrecht Weinert's adaption to weAutSys (and weAutSys — Automation runtime system weAut_01):

- Doxygen documentation
- publishing more symbols and functions (.c/.h re-organising)
- performance improvements for sake of ATMega 8bit µC and ENC28J60
- re-do of call-backs lest to loose valuable DHCP information needed by customer's applications
- requesting (and using) more DHCP options
- re-doing DHCP automatically after renew time set by server answer (using a seconds duration timer)

Besides the impact on dhcp.c and dhcp.h those improvements implied some changes on other uIP files, too.

Files

- file dhcpc.h

  Definitions for the DHCP client.

Data Structures

- struct dhcpMsg_t

  The DHCP message structure.
- struct dhcpOption_t

  The structure of a DHCP message option field.
- struct dhcpState_t

  Structure for DHCP application state.

Defines

- #define BOOTP_BROADCAST 0x8000

  DHCP message flag value.
- #define DHCP_HLEN_ETHERNET 6

  DHCP message hardware address length.
- #define DHCP_HTYPE_ETHERNET 1

  DHCP message type value.
- #define DHCP_OPTION_BROADCAST_ADDR 28

  DHCP option field code: broadcast address.
- #define DHCP_OPTION_CLIENT_NAME 12

  DHCP option field code: host name.
- #define DHCP_OPTION_DEFAULT_TTL 23

  DHCP option field code: default IP TTL.
- #define DHCP_OPTION_DNS_SERVER 6

  DHCP option field code.
- #define DHCP_OPTION_DOMAIN_NAME 15

DHCP option field code: domain name.
- #define DHCP_OPTION_END 255

  DHCP option field code: end of all options.
- #define DHCP_OPTION_LEASE_TIME 51

  The address lease time.
- #define DHCP_OPTION_MSG_TYPE 53

  DHCP option field code: message type.
- #define DHCP_OPTION_NAME_SERVERS 5

  DHCP option field code: name server(s)
- #define DHCP_OPTION_NTP_SERVERS

  DHCP option field code: time server(s)
- #define DHCP_OPTION_REBND_TIME 59

  DHCP option field code.
- #define DHCP_OPTION_RENEW_TIME 58

  DHCP option field code.
- #define DHCP_OPTION_REQ_IPADDR 50

  DHCP option field code.
- #define DHCP_OPTION_REQ_LIST 55

  DHCP request option field code: Parameter Request List.
- #define DHCP_OPTION_ROUTER 3

  DHCP option field code.
- #define DHCP_OPTION_SERVER_ID 54

  DHCP option field code: DHCH server IP address.
- #define DHCP_OPTION_SMTP_SERVERS 69

  DHCP option field code: SMTP server(s)
- #define DHCP_OPTION_SUBNET_MASK 1

  DHCP option field code.
- #define DHCP_OPTION_TIME_OFFSET 2

  DHCP option field code: time offset.
- #define DHCP_OPTION_TIME_SERVERS 4

  DHCP option field code: time server(s)
- #define DHCP_REPLY 2

  DHCP message operation type.
- #define DHCP_REQUEST 1

  DHCP message operation type.
- #define DHCPACK 5

  DHCP message type.
- #define DHCPC_CLIENT_PORT 68

  Used DHCP client port.
- #define DHCPC_SERVER_PORT 67

  Well known DHCP server port.
- #define DHCPDECLINE 4

  DHCP message type.
- #define DHCPDISCOVER 1

  DHCP message type.
- #define DHCPINFORM 8

  DHCP message type.
- #define DHCPNAK 6

  DHCP message type.
- #define DHCPOFFER 2

  DHCP message type.

- #define DHCPRELEASE 7

  DHCP message type.
- #define DHCPREQUEST 3

  DHCP message type.
- #define STATE_CONFIG_RECEIVED 3

  DHCP state machine state.
- #define STATE_INITIAL 0

  DHCP state machine state.
- #define STATE_OFFER_RECEIVED 2

  DHCP state machine state.
- #define STATE_SENDING 1

  DHCP state machine state.

Functions

- ptfnct_t dhcpc_appcall (void)

  Handle DHCP server events.
- void dhcpc_configured (uint8_t respType, const uint16_t ipAddr[ ])

  DHCP success.
- void dhcpcGotOption (const struct dhcpOption_t ∗dhcpOption)

  DHCP option received.
- void dhcpInit (void const ∗mac_addr)

  Initialise the DHCP (client)
- void dhcpReset (void)

  Reset the DHCP (client)

Variables

- struct dhcpState_t dhcpState

  DHCP application state.

2.51.2 Define Documentation

2.51.2.1 #define DHCP_OPTION_MSG_TYPE 53

DHCP option field code: message type.

This is usually (if not always) the first option in a DHCP package. It redundantly gives the message type in form of an option.

The rationale behind is to inform an option parser / option handler in an uniform way about the nature of the package in question.

Especially DHCPACK would mean that all other options are valid and meant to be used by the client.

For the possible options see also RFC2132.

2.51.2.2 #define DHCP_OPTION_TIME_OFFSET 2

DHCP option field code: time offset.

This is the time offset of the client's subnet (location) in seconds relative to UTC. This value can be used to calculate local time from NTP server responses.

2.51.2.3 #define DHCP_OPTION_TIME_SERVERS 4

DHCP option field code: time server(s)

This is a list of 1..n time server addresses with decreasing preference. By DHCP definition those servers use time protocol (RFC 868, port 37) not NTP (RFC 1305, port 123).

See also: DHCP_OPTION_NTP_SERVERS

2.51.2.4 #define DHCP_OPTION_NAME_SERVERS 5

DHCP option field code: name server(s)

This is a list of 1..n name server addresses with decreasing preference.

2.51.2.5 #define DHCP_OPTION_CLIENT_NAME 12

DHCP option field code: host name.

This is to give the client a name by the central organisation.

2.51.2.6 #define DHCP_OPTION_DOMAIN_NAME 15

DHCP option field code: domain name.

This is to give the domain name (sometimes called DNS suffix) that client should use when resolving hostnames via the Domain Name System (DNS). This option can deliver just one sufix, not a list of.

2.51.2.7 #define DHCP_OPTION_DEFAULT_TTL 23

DHCP option field code: default IP TTL.

This is the (default) time to live for outgoing connections.

2.51.2.8 #define DHCP_OPTION_NTP_SERVERS

DHCP option field code: time server(s)

This is a list of 1..n NTP server addresses with decreasing preference. Those servers use NTP (RFC 1305, port 123).

See also: DHCP_OPTION_TIME_SERVERS

2.51.2.9 #define DHCP_OPTION_LEASE_TIME 51

The address lease time.

This option is the requested respectively granted lease time for the address given by the DHCP server in units of seconds (unsigned 32 bit, wrong endian).

Time to renew (T1) is 50% of the lease time by default and may be sent by the server as DHCP_OPTION_RENE-W_TIME option too.

Time to rebind (T2) is 87.5% (7/8) of the lease time by default and may be sent by the server as DHCP_OPTION_-REBND_TIME option too.

2.51.2.10 #define DHCP_OPTION_SERVER_ID 54

DHCP option field code: DHCH server IP address.

This option allows to distinguish several DHCP servers in requests and offers. DHCP option field code

2.51.2.11 #define DHCP_OPTION_SMTP_SERVERS 69

DHCP option field code: SMTP server(s)

This is a list of 1..n server addresses for the Simple Mail Transport Protocol (SMTP) with decreasing preference.

2.51.2.12 #define DHCP_OPTION_END 255

DHCP option field code: end of all options.

This is always the last option in a DHCP package. It is empty and just signals the end of the list. DHCP option field code

2.51.3 Function Documentation

2.51.3.1 void dhcpInit (void const ∗ *mac_addr*)

Initialise the DHCP (client)

This function initialises the DHCP structures and state machine (= protothread) like does dhcpReset. Then it is tried to establish connections to DHCP server and client port(s).

The protocol is not started by this function; the work is done by dhcpc_appcall.

Parameters

mac_addr	pointer to the device's MAC address

2.51.3.2 void dhcpReset (void)

Reset the DHCP (client)

This function resets all DHCP state to initial. In contrast to dhcpInit no trial to connect to DHCP ports is made and the protocol is not (not even indirectly) started.

2.51.3.3 ptfnct_t dhcpc_appcall (void)

Handle DHCP server events.

This function handles DHCP events. And it is a (as a protothread) the DHCP client state machine. It has to be called in the udp_appcall in a manner like

```
const uint16_t remPort = convert16endian(uip_udp_conn->rport);

if(remPort == DHCPC_SERVER_PORT) {  // DHCP protocol
   dhcpc_appcall();
   return;
} // DHCP protocol
```

2.51.3.4 void dhcpcGotOption (const struct dhcpOption_t ∗ *dhcpOption*)

DHCP option received.

This (callback) signals an option send by the DHCP server. Its up to the implementing application or system software to utilise (or ignore) the respective values.

Parameters

dhcpOption	the option (one of the options) received

2.51.3.5 void **dhcpc_configured** (uint8_t *respType*, const uint16_t *ipAddr[ ]*)

DHCP success.

This (callback) signals a successful walk through the DHCP protocol steps ended by acknowledge.

Parameters

respType	at present always DHCPACK; (no callback in other cases yet)
ipAddr	the assigned / leased IP address (in case of success)

2.52 DNS resolver

2.52.1 Overview

The uIP DNS resolver functions are used to lookup a hostname and map it to an IP address. It maintains a list of resolved hostnames that can be queried with the resolv_lookup() function. New hostnames can be resolved using the resolv_query() function.

When a hostname has been resolved (or found to be non-existent), the resolver code calls a callback function called resolv_found() that must be implemented by the module that uses the resolver.

Files

- file resolv.h

  DNS resolver definitions.

Defines

- #define MAX_DNS_RETRIES 8

  The maximum number of DNS retries.

Functions

- void resolv_appcall (void)

  uIP event function for the DNS resolver
- void resolv_conf (uint16_t ∗dnsserver)

  Configure which DNS server to use for queries.
- void resolv_found (char ∗name, uint16_t ∗ipaddr)

  Callback function which is called after hostname look-up.
- uint16_t ∗ resolv_getserver (void)

  Obtain the currently configured DNS server.
- void resolv_init (void)

  Initialise the resolver.
- uint16_t ∗ resolv_lookup (char ∗name)

  Look up a hostname in the array of known hostnames.
- void resolv_query (char ∗name)

  Queues a name so that a question for the name will be sent out.

2.52.2 Function Documentation

2.52.2.1 void **resolv_appcall** (void)

uIP event function for the DNS resolver

Must only be called for the correct protocol / port (rport 53).

2.52.2.2 void **resolv_found** (char ∗ *name,* uint16_t ∗ *ipaddr*)

Callback function which is called after hostname look-up.

This function must be implemented by the module that uses the DNS resolver. It is called when the hostname requested got an IP address assigned by DNS server answer, or when such assignment could not be found.

Parameters

name	pointer to the name that was to look up
ipaddr	pointer to a 4-byte array containing the IP address of the hostname, or NULL if the hostname could not be found.

2.52.2.3 void resolv_conf (uint16_t ∗ *dnsserver*)

Configure which DNS server to use for queries.

Note

weAutSys configuration can hold up to two DNS server IPs usually got by DHCP.

Parameters

dnsserver	a pointer to a 4-byte representation of the DNS server's IP address

2.52.2.4 uint16_t∗ resolv_getserver (void)

Obtain the currently configured DNS server.

Returns

A pointer to a 4-byte representation of the IP address of the currently configured DNS server or NULL if no DNS server has been configured.

2.52.2.5 uint16_t∗ resolv_lookup (char ∗ *name*)

Look up a hostname in the array of known hostnames.

Note

This function only looks in the internal array of known hostnames. It does not send out a query for the hostname, even if none was found. The function resolv_query() can be used to send such query.

Returns

A pointer to a 4-byte representation of the hostname's IP address, or NULL if the hostname was not found in the array of hostnames.

2.52.2.6 void resolv_query (char ∗ *name*)

Queues a name so that a question for the name will be sent out.

Parameters

name	The hostname that is to be queried.

2.53 + + fatFS -- ChaN's FAT filesystem implementation + +

2.53.1 Overview

ChaN contributed two FAT file system implementations as open source

- a small variant Petit FatFs and
- a quite bigger (Grande en lieu de petit?) full featured one just called fatFS

Both are great work and put under a license that allowed use in commercial software.

For weAutSys / weAut_01 the petit one was unsuitable restricted. And it proved quite hard to upgrade. So the grande variant was taken and cut to size quite radically for (mainly) two reasons

1. adaption of ChaN's powerful, flexible and enormous implementation to the comparatively minikin ATmega processor
2. cutting the functions and procedures long running on slow (10 MHz) SPI interfaced SMCs in pieces compatible to a no-preemptive real time OS </li/

The successful adaption to weAutSys / weAut_01 respectively ATmega is still open source. It, necessarily, sacrifices a good deal of ChaN's original flexibility and adaptability to large systems with multiple drives of many types.

Modules

- Driver adaption to SMC
- File system operations

Files

- file diskio.h

  Small memory card (SMC / MMC) driver function definitions.
- file ff.h

  (Grande) fatFS configurations and declarations
- file ffconf.h

  (Grande) fatFS configuration options

Defines

- #define _CODE_PAGE 1

  The code page used.
- #define _FS_MINIMIZE 0

  Omit functions to save (program / flash) memory.
- #define _FS_READONLY 0

  Only read functions.
- #define _FS_RPATH 0

  The relative path feature.
- #define _LFN_UNICODE 0

  0: ANSI/OEM or 1: Unicode
- #define _MAX_LFN 255

  Maximum LFN length to handle (12 to 255)
- #define _MULTI_PARTITION 0

Allow multiple partitions.
- #define _USE_ERASE 0

  Allow erase of blocks of sectors.
- #define _USE_FASTSEEK 0

  0: disable or 1: enable the fast seek feature
- #define _USE_LFN 0

  The long file name feature.
- #define _USE_STRFUNC 0

  0: disable or 1-2: enable string functions
- #define _VOLUMES 1

  Number of volumes (logical drives) to be used.
- #define _WORD_ACCESS 1

  Word access.

2.53.2 Define Documentation

2.53.2.1 #define _FS_READONLY 0

Only read functions.

In a read only configuration all write functions would be removed: f_write, f_sync, f_unlink, f_mkdir, f_chmod, f_-rename, f_truncate as well as f_getfree (then useless).

Values: 0 = read/write or 1 = read only

2.53.2.2 #define _FS_MINIMIZE 0

Omit functions to save (program / flash) memory.

The _FS_MINIMIZE option defines minimisation level to remove some functions.

0: No reduction, except on weAutSys omission of f_getfree and f_lseek ∗a)

1: Omit f_stat, f_getfree, f_unlink, f_mkdir, f_chmod, f_truncate and f_rename.

2: Additionally omit f_opendir and f_readdir.

3: Omit f_lseek also.

Note ∗a): f_getfree and f_lseek follow sectors until certain conditions are met. This leads to unpredictable run times — often long enough to fire a watchdog reset.

2.53.2.3 #define _CODE_PAGE 1

The code page used.

The _CODE_PAGE specifies the OEM code page to be used on the target system. Incorrect setting of the code page can cause a file open failure.

932 - Japanese Shift-JIS (DBCS, OEM, Windows)

936 - Simplified Chinese GBK (DBCS, OEM, Windows)

949 - Korean (DBCS, OEM, Windows)

950 - Traditional Chinese Big5 (DBCS, OEM, Windows)

1250 - Central Europe (Windows)

1251 - Cyrillic (Windows)

1252 - Latin 1 (Windows)

1253 - Greek (Windows)

1254 - Turkish (Windows)

1255 - Hebrew (Windows)

1256 - Arabic (Windows)

1257 - Baltic (Windows)

1258 - Vietnam (OEM, Windows)

437 - U.S. (OEM)

720 - Arabic (OEM)

737 - Greek (OEM)

775 - Baltic (OEM)

850 - Multilingual Latin 1 (OEM)

858 - Multilingual Latin 1 + Euro (OEM)

852 - Latin 2 (OEM)

855 - Cyrillic (OEM)

866 - Russian (OEM)

857 - Turkish (OEM)

862 - Hebrew (OEM)

874 - Thai (OEM, Windows)

1 - ASCII only (Valid for non LFN configuration)

2.53.2.4 #define _USE_LFN 0

The long file name feature.

0: Disable LFN feature. _MAX_LFN and _LFN_UNICODE have no effect.

1: Enable LFN with static working buffer on the BSS. Always NOT reentrant.

2: Enable LFN with dynamic working buffer on the stack.

3: Enable LFN with dynamic working buffer on the heap.

The LFN working buffer occupies (_MAX_LFN + 1) $*$ 2 bytes. To enable LFN, the Unicode handling functions ff_-convert() and ff_wtoupper() must be added to the project. To use heap, the memory control functions ff_memalloc() and ff_memfree() must be added to the project.

For small systems and SMCs this feature is switched off for space, performance and license reasons. (Microsoft holds patents for long file names on FAT file systems.) One extra drawback of using only short file names is their using only upper case letters.

2.53.2.5 #define _FS_RPATH 0

The relative path feature.

0: Disable relative path feature and remove related functions.

1: Enable relative path functions f_chdrive() and f_chdir().

2: f_getcwd() is available, also.

Note that output of f_readdir() is affected by this option.

For small systems and SMCs this feature is switched off.

2.53.2.6 #define _MULTI_PARTITION 0

Allow multiple partitions.

legal values:

0: single partition,

1/2: enable multiple partition

When set to 0, each volume is bound to the same physical drive number and can mount only the first primary partition. When it is set to 1 or 2, each volume is tied to the partitions listed in (then defined) VolToPart[]. When set to 2 a f_fisk() function is defined additionally.

Hint: For small memory card (SMC, MMC) implementation on small controllers _MULTI_PARTITION should be kept 0.

2.53.2.7 #define _USE_ERASE 0

Allow erase of blocks of sectors.

This is a device driver function "below" the file system. For use with weAutSys it should be kept disabled to avoid conflicts with the runtime's driver functions.

Note: Files and directories are erased by f_unlink().

0: disable / 1: enable sectors erase.

If set to 1 the CTRL_ERASE_SECTOR command should be added to the disk_ioctl function.

2.53.2.8 #define _WORD_ACCESS 1

Word access.

values: 0: Byte-by-byte access / 1: Word access.

ChaN says: When the byte order on the memory is big-endian or address miss-aligned word access results incorrect behavior, the _WORD_ACCESS must be set to 0.

Hint (A.W.): In all other cases (like ref intro_sec "weAutSys", ref intro_secH "weAut_01", ATmega, GCC) it should be left 1 as the C and machine code generated by `_WORD_ACCESS` == 0 is not optimal (to put it mildly).

Hint2: The name of this configuration option is a bit misleading. It concerns not really the bus width (8, 16, 32, 64) of the computer architecture used but more (or only) its endianess.

2.54 Driver adaption to SMC

2.54.1 Overview

The functions in this architecture specific module mainly implement the glue code between ChaN's file system adaption and weAutSys' driver and handling functions for small memory cards (SMCs).

Defines

- #define _USE_IOCTL 1

  enable disk_ioctl function
- #define _USE_WRITE 1

  enable disk_write function
- #define CT_BLOCK 0x08

  Card type flag: block addressing.
- #define CT_MMC 0x01

  Card type flag: MMC version 3.
- #define CT_SD1 0x02

  Card type flag: SD version 1.
- #define CT_SD2 0x04

  Card type flag: SD version 2.
- #define CT_SDC

  Card type mask: SD.
- #define CTRL_ERASE_SECTOR 4

  Generic command: force erased a block of sectors (for only _USE_ERASE)
- #define CTRL_POWER 5

  Generic command: get/set power status.
- #define DRESULT

  Results of disk (media driver) functions.
- #define GET_BLOCK_SIZE 3

  Generic command: get erase block size (for only f_mkfs())
- #define GET_SECTOR_COUNT 1

  Generic command: get media size (for only f_mkfs())
- #define GET_SECTOR_SIZE 2

  Generic command: get sector size.
- #define MMC_GET_CID 12

  Specific ioctl command: get CID.
- #define MMC_GET_CSD 11

  Specific ioctl command: get CSD.
- #define MMC_GET_OCR 13

  Specific ioctl command: get OCR.
- #define MMC_GET_SDSTAT 14

  Specific ioctl command: get SMC status.
- #define MMC_GET_TYPE 10

  Specific ioctl command: get card type.
- #define RES_ERROR 1

  Any (hard) error occurred.
- #define RES_NOTRDY 3

  The drive has not been initialised.
- #define RES_PARERR 3

Invalid function argument.

- #define STA_NODISK 0x02

  No medium in the drive / slot.
- #define STA_NOINIT 0x01

  Drive not initialized.
- #define STA_PROTECT 0x04

  Medium is write protected by hardware switch.

Functions

- uint8_t disk_initialize (uint8_t drv)

  Initialise the SMC drive.
- DRESULT disk_ioctl (uint8_t drv, uint8_t ctrl, uint8_t ∗buff)

  Special control functions.
- DRESULT disk_read (uint8_t drv, uint8_t ∗buff, uint32_t sector)

  Read a sector.
- uint8_t disk_status (uint8_t drv)

  Get drive status.
- DRESULT disk_write (uint8_t drv, const uint8_t ∗buff, uint32_t sector)

  Write a sector.

2.54.2 Define Documentation

2.54.2.1 #define DRESULT

Results of disk (media driver) functions.

Hint: The values assigned here for (Grande) FAT FS are not the same as with Petit FatFs. Remark 2: Not any usage of the (three) media access functions within this fatFS version really uses those values: Some just compare to 0 (RES_OK) and some do not care at all. Hence we (this weAutSys port, A.W.) change it to uint8_t and "0 = false = no error = ".

2.54.2.2 #define STA_NOINIT 0x01

Drive not initialized.

This status (mask) bit indicates that the disk drive has not been initialised. The type of the (inserted) SMC is not determined yet .

See also

STA_NODISK
disk_status()

2.54.2.3 #define STA_NODISK 0x02

No medium in the drive / slot.

This status (mask) bit indicates no medium being inserted in the drive respectively in the small memory card (SMC / MMC) connector.

See also

STA_NOINIT
disk_status()
smcInsertSwitch()
smcInsPow()

2.54.2.4 #define STA_PROTECT 0x04

Medium is write protected by hardware switch.

Small memory cards (SMCs) do not have a write protect slider and the MMC-slot built in weAut_01 accordingly has no WP-switch. Hence this feature is not implemented and this status won't occur.

See also

STA_NOINIT
STA_NODISK

2.54.3 Function Documentation

2.54.3.1 uint8_t disk_initialize (uint8_t *drv*)

Initialise the SMC drive.

This function is used in fatFS to initialise the drive. Legal return values are STA_NOINIT, STA_NODISK, STA_PROTECT and 0 for OK. STA_PROTECT cannot occur with SMCs as there is no hardware write protect mechanism.

With weAutSys this function just checks the initialisation, hence doing (in effect) same as disk_status(). The reason is weAutSys doing the SMC initialisation and other time consuming SMC actions in a background (Protothread) thread .

Parameters

drv	the number of the physical drive to initialise (must be 0 in this implementation)

Returns

0: success, STA_NOINIT | STA_NODISK

2.54.3.2 uint8_t disk_status (uint8_t *drv*)

Get drive status.

This function just informs about the drive status without actually accessing the device.

Parameters

drv	the physical drive number to check (must be 0 in this implementation)

Returns

0: success, STA_NOINIT | STA_NODISK

See also

disk_initialize()

2.54.3.3 DRESULT disk_read (uint8_t *drv,* uint8_t ∗ *buff,* uint32_t *sector*)

Read a sector.

This function reads a sector into the buffer `buff` supplied. As the process takes 2 ms on SMCs with 5MHz SPI clock frequency multiple sectors are disallowed. This function basically delegates to readDataBlock(sector, buff). Hence the sector number `sector` is just that cause weAutSys's readDataBlock() takes care of address transformations required for some low capacity SMCs.

Instead of doing block operations by this functions it is preferable to let weAutSys handle that in the respective background thread. That would then be done more compatibly with real time (process automation) tasks.

The same restrictions and recommendations apply to disk_write().

Parameters

drv	the physical drive number to initialise (must be 0 in this implementation)
buff	Pointer to the read buffer (NULL not allowed)
sector	Sector number (LBA)

2.54.3.4 DRESULT disk_write (uint8_t *drv,* const uint8_t ∗ *buff,* uint32_t *sector*)

Write a sector.

In the sense described with disk_read() this function basically delegates to writeDataBlock(sector, buff).

Parameters

drv	the physical drive number to initialise (must be 0 in this implementation)
buff	pointer to the write buffer (NULL: not allowed)
sector	Sector number (LBA)

2.54.3.5 DRESULT disk_ioctl (uint8_t *drv,* uint8_t *ctrl,* uint8_t ∗ *buff*)

Special control functions.

Depending on the device and implementation this function allows to get (/put) certain information from the device:

CTRL_POWER : no effect on weAut_01 module

GET_SECTOR_COUNT : get number of sectors on the disk (uint32_t)

GET_SECTOR_SIZE : get R/W sector size (uint16_t; this implementation will always return 512)

GET_BLOCK_SIZE : get the erase block size in unit of sectors (uint16_t)

MMC_GET_TYPE : get card type flags (1 byte)

MMC_GET_CSD : receive CSD as a data block (16 bytes)

MMC_GET_CID : receive CID as a data block (16 bytes)

MMC_GET_OCR : receive OCR as an R3 (4 bytes)

MMC_GET_SDSTAT : receive SD status as a data block (64 bytes)

Hint: This function is not implemented if _USE_IOCTL is undefined or 0.

Parameters

drv	the physical drive number to initialise (must be 0 in this implementation)
ctrl	control code
buff	buffer to send/receive control data

2.55 File system operations

2.55.1 Overview

The file operations provided by ChaN's fatFS were adapted to weAut_01 like hardware, the use with one slot for small memory cards (SMCs) and the weAutSys runtime. The necessary "cut to size" was quite far-reaching. But it was not so radical, that the restriction to one device and one file system could not be lifted up easily for future upgrading.

Data Structures

- struct DIR

  Directory object structure (DIR)
- struct FATFS

  File system object structure (FATFS)
- struct FIL

  File object structure (FIL)
- struct FILINFO

  File status structure (FILINFO)

Defines

- #define _FS_TINY

  Tiny FS.
- #define _MAX_SS 512

  Maximum sector size to be handled.
- #define AM_ARC 0x20

  directory entry attribute bit: archive
- #define AM_DIR 0x10

  directory entry attribute bit: directory
- #define AM_HID 0x02

  directory entry attribute bit: hidden
- #define AM_LFN 0x0F

  directory entry attribute bit: LFN entry
- #define AM_MASK 0x3F

  mask of defined directory entry attribute bits
- #define AM_RDO 0x01

  directory entry attribute bit: read only
- #define AM_SYS 0x04

  directory entry attribute bit: system
- #define AM_VOL 0x08

  directory entry attribute bit: Volume label
- #define div16SSZ(div)

  Divide (16 bit, unsigned) by fixed sector size.
- #define div2SSZ(div)

  Divide (32 bit, unsigned) by 2 times fixed sector size.
- #define div32SSZ(div)

  Divide (32 bit, unsigned) by fixed sector size.
- #define div4SSZ(div)

  Divide (32 bit, unsigned) by 4 times fixed sector size.
- #define divSSZ2(div)

Divide (32 bit, unsigned) by fixed sector size by 2.

- #define divSSZ4(div)

  Divide (32 bit, unsigned) by fixed sector size by 4.
- #define f_eof(fp)

  Check EOF.
- #define f_error(fp)

  Check error state.
- #define f_size(fp)

  Get the size.
- #define f_tell(fp)

  Get the current read/write pointer.
- #define FA_CREATE_ALWAYS 0x08

  Mode flag: create anyway.
- #define FA_CREATE_NEW 0x04

  Mode flag: create a new file.
- #define FA_OPEN_ALWAYS 0x10

  Mode flag: open anyway.
- #define FA_OPEN_EXISTING 0

  Mode flag: open the existing file object.
- #define FA_READ 0x01

  Mode flag: read access to the file object.
- #define FA_WRITE 0x02

  Mode flag: write access to the file object.
- #define FS_FAT12 1

  FATFS.fs_type value.
- #define FS_FAT16 2

  FATFS.fs_type value.
- #define FS_FAT32 3

  FATFS.fs_type value.
- #define get_fattime()

  Get current local FAT time.
- #define LD2PD(vol)

  Each logical drive is bound to the same physical drive number ∗/.
- #define LD2PT(vol)

  Always mounts the 1st partition or in SFD.
- #define modSSZ(div)

  Modulo (unsigned) by fixed sector size.
- #define mul16SSZ(fac)

  Multiply (16 bit, unsigned) with fixed sector size.
- #define mul32SSZ(fac)

  Multiply (32 bit, unsigned) with fixed sector size.
- #define SSM 511

  Mask for fixed sector size.
- #define SSZ 512

  Fixed sector size.

Enumerations

- enum FRESULT {
  FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY,
  FR_NO_FILE, FR_NO_PATH, FR_INVALID_NAME, FR_DENIED,
  FR_EXIST, FR_INVALID_OBJECT, FR_WRITE_PROTECTED, FR_INVALID_DRIVE,
  FR_NOT_ENABLED, FR_NO_FILESYSTEM, FR_MKFS_ABORTED, FR_TIMEOUT,
  FR_LOCKED, FR_NOT_ENOUGH_CORE, FR_TOO_MANY_OPEN_FILES, FR_INVALID_PARAMETER
  }

  File functions return code.

Functions

- FRESULT checkValidFS (uint8_t vol)

  Check if there's a valid mounted file system on the drive.
- FRESULT checkWriteProtect (uint8_t vol)

  Check if the drive is write protected.
- uint8_t extractDrive (const TCHAR ∗∗path)

  Extract drive number from path.
- FRESULT f_chdir (const TCHAR ∗)

  Change current directory (not implemented)
- FRESULT f_chdrive (uint8_t)

  Change current drive (not implemented)
- FRESULT f_chmod (const TCHAR ∗path, uint8_t value, uint8_t mask)

  Change attribute of the file or directory.
- FRESULT f_close (FIL ∗fp)

  Close an open file object.
- FRESULT f_fdisk (uint8_t pdrv, const uint32_t szt[ ], void ∗work)

  Divide a physical drive into some partitions.
- FRESULT f_getcwd (TCHAR ∗, uint16_t)

  Get current directory (not implemented)
- FRESULT f_getfree (const uint8_t vol, uint32_t ∗nclst)

  Get the number of free clusters on the drive.
- TCHAR ∗ f_gets (TCHAR ∗, int, FIL ∗)

  Get a string from the file (not implemented)
- FRESULT f_lseek (FIL ∗fp, uint32_t ofs)

  Move file pointer of a file object, expand file size.
- FRESULT f_mkdir (const TCHAR ∗path)

  Create a new directory.
- FRESULT f_mount (uint8_t vol, FATFS ∗fs)

  Mount / unmount a logical drive.
- FRESULT f_open (FIL ∗fp, const TCHAR ∗path, uint8_t mode)

  Open or create a file.
- FRESULT f_opendir (DIR ∗dj, const TCHAR ∗path)

  Open an existing directory.
- int f_printf (FIL ∗, const TCHAR ∗,...)

  Put a formatted string to the file (not implemented)
- int f_putc (TCHAR, FIL ∗)

  Put a character to the file (not implemented)
- int f_puts (const TCHAR ∗, FIL ∗)

  Put a string to the file (not implemented)

- FRESULT f_read (FIL ∗fp, void ∗buff, uint16_t btr, uint16_t ∗br)

  Read data from a file.
- FRESULT f_readdir (DIR ∗dj, FILINFO ∗fno)

  Read a directory item.
- FRESULT f_rename (const TCHAR ∗path_old, const TCHAR ∗path_new)

  Rename or move a file or directory.
- FRESULT f_stat (const TCHAR ∗path, FILINFO ∗fno)

  Get file status.
- FRESULT f_sync (FIL ∗fp)

  Flush written / cached data to a file.
- FRESULT f_truncate (FIL ∗fp)

  Truncate file.
- FRESULT f_unlink (const TCHAR ∗path)

  Delete an existing file or directory.
- FRESULT f_utime (const TCHAR ∗path, const FILINFO ∗fno)

  Change time-stamps of a file or directory.
- FRESULT f_write (FIL ∗fp, const void ∗buff, uint16_t btw, uint16_t ∗bw)

  Write data to a file.
- FRESULT getValidFS (uint8_t vol, FATFS ∗∗fatfs)

  Get the file system for the drive.
- FRESULT lookForFS (uint8_t vol)

  Look for a recognised file system on the drive.

2.55.2 Define Documentation

2.55.2.1 #define _MAX_SS 512

Maximum sector size to be handled.

Always set 512 for memory card and hard disk but a larger value may be required for on-board flash memory, floppy disk and optical disk. When _MAX_SS is larger than 512, it configures FatFs to variable sector size and GET_SECTOR_SIZE command must be implemented by the disk_ioctl() function.

Hint: For small memory card (SMC, MMC) implementation GET_SECTOR_SIZE is implemented in disk_ioctl() blindly returning 512.

legal values for original fatFS: 512, 1024, 2048 or 4096 (configurable)

fixed value for this implementation: 512

2.55.2.2 #define SSZ 512

Fixed sector size.

See also

  _MAX_SS

2.55.2.3 #define mul32SSZ(*fac*)

Multiply (32 bit, unsigned) with fixed sector size.

See also

  SSZ

Parameters

fac	the 32 bit (unsigned) factor fac's upper 9 bits are ignored and should be 0 to get a arithmetically correct result

Returns

the product fac ∗ SSZ

See also

div32SSZ(div)

2.55.2.4 #define mul16SSZ(*fac*)

Multiply (16 bit, unsigned) with fixed sector size.

See also

SSZ

Parameters

fac	the 16 bit (unsigned) factor fac's upper 9 bits are ignored and should be 0 to get a arithmetically correct result

Returns

the product fac ∗ SSZ

See also

mul32SSZ(div)

2.55.2.5 #define div32SSZ(*div*)

Divide (32 bit, unsigned) by fixed sector size.

See also

SSZ

Parameters

div	the 32 bit (unsigned) dividend

Returns

the quotient div / SSZ

See also

mul32SSZ(fac)
div2SSZ(div)
div4SSZ(div)

2.55.2.6 #define div16SSZ(*div*)

Divide (16 bit, unsigned) by fixed sector size.

See also

SSZ

Parameters

div	the 16 bit (unsigned) dividend

Returns

the quotient div / SSZ

See also

mul32SSZ(fac)
div32SSZ(div)

2.55.2.7 #define modSSZ(*div*)

Modulo (unsigned) by fixed sector size.

See also

SSZ

Parameters

div	the (unsigned) dividend

Returns

the remainder div % SSZ

See also

mul32SSZ(fac)

2.55.2.8 #define div2SSZ(*div*)

Divide (32 bit, unsigned) by 2 times fixed sector size.

See also

SSZ

Parameters

div	the 32 bit (unsigned) dividend

Returns

the quotient div / (SSZ $*$ 2)

See also

mul32SSZ(fac)

2.55.2.9 #define div4SSZ(*div*)

Divide (32 bit, unsigned) by 4 times fixed sector size.

See also

SSZ

Parameters

div	the 32 bit (unsigned) dividend

Returns

the quotient div / (SSZ $*$ 4)

See also

mul32SSZ(fac)

2.55.2.10 #define divSSZ4(*div*)

Divide (32 bit, unsigned) by fixed sector size by 4.

This divides a 32 bit number by a quarter of the fixed sector size.

See also

SSZ

Parameters

div	the 32 bit (unsigned) dividend

Returns

the quotient div / (SSZ / 4)

See also

mul32SSZ(fac)

2.55.2.11 #define divSSZ2(*div*)

Divide (32 bit, unsigned) by fixed sector size by 2.

This divides a 32 bit number by a quarter of the fixed sector size.

See also

SSZ

Parameters

div	the 32 bit (unsigned) dividend

Returns

the quotient div / (SSZ / 2)

See also

mul32SSZ(fac)

2.55.2.12 #define _FS_TINY

Tiny FS.

When _FS_TINY is set to 1, fatFS uses the sector buffer in the file system object instead of the sector buffer in the individual file object for file data transfer. This reduces the memory consumption by 512 bytes for each file object.

Legal values on original fatFS: 0 = normal or 1 = tiny — configurable there but fixed for this implementation.

2.55.2.13 #define f_eof(*fp*)

Check EOF.

Parameters

fp	Pointer to the file's object structure (type FIL)

Returns

1 if EOF reached

2.55.2.14 #define f_error(*fp*)

Check error state.

Parameters

fp	Pointer to the file's object structure (type FIL)

Returns

1 if in an erroneous state

2.55.2.15 #define f_tell(*fp*)

Get the current read/write pointer.

Parameters

fp	Pointer to the open file's object structure (type FIL)

Returns

the current R/W pointer, i.e. byte index (type uint32_t)

2.55.2.16 #define f_size(*fp*)

Get the size.

Parameters

fp	Pointer to the open file's object structure (type FIL)

Returns

the size (type uint32_t)

2.55.2.17 #define get_fattime( )

Get current local FAT time.

See also

getFATtime

2.55.2.18 #define FA_READ 0x01

Mode flag: read access to the file object.

Combine with FA_WRITE for read-write access.

Examples:

main.c.

2.55.2.19 #define FA_OPEN_EXISTING 0

Mode flag: open the existing file object.

The function fails if the file is not existing (default).

Examples:

main.c.

2.55.2.20 #define FA_WRITE 0x02

Mode flag: write access to the file object.

Combine with FA_READ for read-write access.

2.55.2.21 #define FA_CREATE_NEW 0x04

Mode flag: create a new file.

The function fails with FR_EXIST if the file is existing.

2.55.2.22 #define FA_CREATE_ALWAYS 0x08

Mode flag: create anyway.

Opens the file if it is existing. If not, a new file is created.

2.55.2.23 #define FA_OPEN_ALWAYS 0x10

Mode flag: open anyway.

Opens the file if it is existing. If not, a new file is created. To append data to the file, use f_lseek after file opening with this mode.

2.55.3 Enumeration Type Documentation

2.55.3.1 enum FRESULT

File functions return code.

Enumerator:

FR_OK (0) Succeeded

FR_DISK_ERR (1) A hard error occurred in the low level disk I/O layer

FR_INT_ERR (2) Assertion failed

FR_NOT_READY (3) The physical drive cannot work

FR_NO_FILE (4) Could not find the file

FR_NO_PATH (5) Could not find the path

FR_INVALID_NAME (6) The path name format is invalid

FR_DENIED (7) Access denied due to prohibited access or directory full

FR_EXIST (8) Access denied due to prohibited access as already existing

FR_INVALID_OBJECT (9) The file/directory object is invalid

FR_WRITE_PROTECTED (10) The physical drive is write protected

FR_INVALID_DRIVE (11) The logical drive number is invalid

FR_NOT_ENABLED (12) The volume has no work area

FR_NO_FILESYSTEM (13) There is no valid FAT volume

FR_MKFS_ABORTED (14) The f_mkfs() aborted due to any parameter error

FR_TIMEOUT (15) Could not get a grant to access the volume within defined period

FR_LOCKED (16) The operation is rejected according to the file sharing policy

FR_NOT_ENOUGH_CORE (17) LFN working buffer could not be allocated

FR_TOO_MANY_OPEN_FILES (18) Number of open files $>$ _FS_SHARE

FR_INVALID_PARAMETER (19) Given parameter is invalid

2.55.4 Function Documentation

2.55.4.1 FRESULT f_mount (uint8_t *vol,* FATFS ∗ *fs*)

Mount / unmount a logical drive.

This function registers / unregisters a work area, i.e. FATFS structure, assigning it to a drive number (0...) respectively volume. A work area must be assigned to the a volume before the use of any other file function. To unregister a work area use NULL as parameter `fs`.

This function always succeeds for a legal `vol` value regardless of the drive's status. No media is accessed by this function. It only clears the given work area and registers it.

The "real" volume mount process is performed

1. once on first file access after calling f_mount or
2. once on first file access after after a media change.

Parameters

vol	Logical drive number to be mounted / unmounted
fs	Pointer to new file system object (NULL for unmount only)

Returns

FR_OK, FR_INVALID_DRIVE

See also

fileSystSMC
fMountSMC()

2.55.4.2 uint8_t extractDrive (const TCHAR ∗∗ *path*)

Extract drive number from path.

This function checks if `**path` points to character sequence starting with "x:" where x may be '0..9', 'a..j' or 'A..J'. If so, `*path` is incremented by 2 (to point past the :) and 0.. 9 is returned according to the first character. Otherwise 0 is returned (respectively the current drive if enabled).

Parameters

path	Pointer to pointer to the path name ([d:]path)

Returns

drive / volume number in the range 0..9 (default 0)

2.55.4.3 FRESULT checkValidFS (uint8_t *vol*)

Check if there's a valid mounted file system on the drive.

This function checks if drive number `vol` is ready and has a valid recognised file system. In that case FR_OK is returned and file system operations can be used without further preparations or delays.

FR_NO_FILESYSTEM means, of course, no file system has been detected yet. That means it has either not yet been looked for (and all else is OK) or the medium is not formatted with a file system usable by this fatFS implementation.

FR_INVALID_DRIVE means no valid volume number.

FR_NOT_ENABLED means no FS structure assigned.

FR_NOT_READY means drive down or medium removed

Parameters

vol	drive number 0.. (at present only 0 is valid)

Returns

FR_OK (0); FR_INVALID_DRIVE FR_NOT_ENABLED FR_NO_FILESYSTEM FR_NOT_READY

See also

getValidFS

2.55.4.4 FRESULT getValidFS (uint8_t *vol*, FATFS ** *fatfs*)

Get the file system for the drive.

Exactly like checkValidFS() this function checks if drive number `vol` is ready and has a valid recognised file system. In that sense getValidFS() is a substitute for checkValidFS().

Additionally it returns the (pointer to the) file system structure assigned with drive `vol` in parameter `fatfs` if that is possible.

Parameters

vol	drive number 0.. (at present only 0 is valid)
fatfs	Pointer to pointer to the file system object to return

Returns

FR_OK (0); FR_INVALID_DRIVE FR_NOT_ENABLED FR_NO_FILESYSTEM FR_NOT_READY

See also

checkValidFS()

2.55.4.5 FRESULT checkWriteProtect (uint8_t *vol*)

Check if the drive is write protected.

This function checks if drive number `vol` is writable. In that case FR_OK is returned. This result only makes sense it / respectively requires that the drive is been checked / initialised as usable. Otherwise FR_NOT_READY may be returned.

For a usable but write protected drive and file FR_WRITE_PROTECTED will be returned. At present this will not happen on weAutSys / weAut_01 Communication with a small memory card (via SPI) SMC implementations as those slots do not have write protect switches. Nevertheless later versions may add write protect as software feature.

Parameters

vol	drive number 0.. (at present only 0 is valid)

Returns

FR_OK (0); FR_NOT_READY FR_WRITE_PROTECTED

2.55.4.6 FRESULT lookForFS (uint8_t *vol*)

Look for a recognised file system on the drive.

This function checks if drive number `vol` is ready and looks for a valid recognised file system on it. If that is or was discovered FR_OK is returned and file system operations can be used without further preparations or delays.

FR_NO_FILESYSTEM means all else is OK but no file system can be detected on that drive respectively medium. The only remedy (on present weAutSys / weAut_01 Communication with a small memory card (via SPI) SMC implementations) is to remove the medium and format it with a suitable file system (preferably FAT32).

FR_INVALID_DRIVE means no valid volume number.

FR_NOT_ENABLED means no FS structure assigned.

FR_NOT_READY means drive down or medium removed

Parameters

vol	drive number 0.. (at present only 0 is valid)

Returns

FR_OK (0); FR_INVALID_DRIVE FR_NOT_ENABLED FR_NO_FILESYSTEM FR_NOT_READY

2.55.4.7 FRESULT f_open (FIL ∗ *fp*, const TCHAR ∗ *path*, uint8_t *mode*)

Open or create a file.

If this function f_open function succeeded, the file object is valid. The file object is used for subsequent read/write functions to identify the file. To close an open file use f_close(). If the modified file is not closed, the file data can be corrupted.

Before using any file function, a work area (file system object) must be registered to the logical drive with f_mount(). All functions except f_fdisk function can work after this procedure.

The parameter mode can be ORed using the predefined mode flags:

- FA_READ Specifies read access to the object. Data can be read from the file. Combine with FA_WRITE for read-write access.
  - FA_WRITE Specifies write access to the object. Data can be written to the file. Combine with FA_READ for read-write access.
  - FA_OPEN_EXISTING (default) Opens the file if it is existing. Otherwise the function fails.
  - FA_OPEN_ALWAYS Opens the file if it is existing. If not, a new file is created. To append data to the file, use f_lseek() after the file open in this method.
  - FA_CREATE_NEW Creates a new file. The function fails with FR_EXIST if the file is existing.
  - FA_CREATE_ALWAYS Creates a new file. If the file is existing, it is truncated and overwritten.

Parameters

fp	Pointer to the blank file object
path	Pointer to the file name
mode	Access mode and file open mode flags

Returns

FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_NO_FILE, FR_NO_PATH, FR_INVALID_NAME, FR_DENIED, FR_EXIST, FR_WRITE_PROTECTED, FR_INVALID_DRIVE, FR_NOT_ENABLED, FR_NO_FILESYSTEM, FR_TIMEOUT, FR_LOCKED, FR_NOT_ENOUGH_CORE, FR_TOO_MANY_OPEN_FILES

Examples:

main.c.

2.55.4.8 FRESULT f_read (FIL ∗ *fp,* void ∗ *buff,* uint16_t *btr,* uint16_t ∗ *br*)

Read data from a file.

This function reads file data to the buffer supplied. The file object's file pointer increases by the number of bytes read. After the function succeeded, `*br` should be checked to detect the end of file. In the case of `*br` < `btr` the read/write pointer reached end of the file during the read operation.

Parameters

fp	Pointer to file object
buff	Pointer to data buffer
btr	Number of bytes to read
br	Pointer to result: number of bytes read This number must not be greater than the sector size (512)

Returns

FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_INVALID_OBJECT, FR_TIMEOUT

Examples:

main.c.

2.55.4.9 FRESULT f_close (FIL ∗ *fp*)

Close an open file object.

This function closes an open file. If any data have been written to the file, the cached information is written back to the medium (SMC). After the function succeeded, the file object is no longer valid.

Parameters

fp	Pointer to file object

Returns

FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_INVALID_OBJECT, FR_TIMEOUT

Examples:

main.c.

2.55.4.10 FRESULT f_lseek (FIL ∗ *fp,* uint32_t *ofs*)

Move file pointer of a file object, expand file size.

This function moves the file read/write pointer of an open file. The offset is absolute, i.e. from file origin resp. top of file. When an offset above the file size is specified in write mode, the file size is increased and the data in the expanded area are undefined. Thereby is is possible to create a large file quickly. After the f_lseek function succeeded, current read/write pointer should be checked to make sure the read/write pointer has been moved correctly. In case of the current read/write pointer is not the expected value, either of followings has occured:

- End of file. The specified Offset was clipped at end of the file because the file has been opened in read-only mode.
- Disk full. There is insufficient free space on the volume to expand the file size.

Fast seek feature is enabled when _USE_FASTSEEK is set to 1 and the member .cltbl in the file object is not NULL. This feature enables fast backward/long seek operations without FAT access by cluster link map table (CLMT) stored in the user defined table. It is also applied to the functions f_read() and f_write(). In this mode, the file size cannot be increased by f_write() or f_lseek(). The CLMT must be created in the user defined DWORD array prior to use fast seek feature. To create the CLMT, let the member .cltbl in the file object pointer to the DWORD array. Set the array size in unit of items into the first item and call the f_lseek function with Offset = CREATE_LINKMAP. After the function succeeded and CLMT is created, no FAT access will occur in subsequent f_read(), f_write() or f_lseek() calls on that file. If the function failed with FR_NOT_ENOUGH_CORE, the given array size is insufficient for the file and the required items is returned into the first item of the array. The required array size is (number of fragments + 1) $*$ 2 items. For example, when the file is fragmented in 5, 12 items will be required for the CLMT.

Parameters

fp	Pointer to the file object
ofs	File pointer from top of file

Returns

FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_INVALID_OBJECT, FR_TIMEOUT

2.55.4.11 FRESULT f_opendir (DIR $*$ *dj*, const TCHAR $*$ *path*)

Open an existing directory.

This function opens an existing directory and creates the directory object for subsequent calls of directory related functions. The directory object can be discarded at any time.

Parameters

path	Pointer to the directory's file name (0-terminated)
dj	Pointer to directory object to create

Returns

FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_NO_PATH, FR_INVALID_NAME, FR_INVALID_DRIVE, FR_NOT_ENABLED, FR_NO_FILESYSTEM, FR_TIMEOUT, FR_NOT_ENOUGH_CORE

2.55.4.12 FRESULT f_readdir (DIR $*$ *dj*, FILINFO $*$ *fno*)

Read a directory item.

This function reads directory entries in sequence. All items in the directory can be read by repeated calls. When all directory entries have been thus visited the function just puts a null string into fno.f_name[]. When a null pointer is given to the FileInfo, the read index of the directory object will be rewound.

When LFN feature is enabled, lfname and lfsize in the file information structure must be initialized with valid value prior to use the f_readdir function. The lfname is a pointer to the string buffer to return the long file name. The lfsize is the size of the string buffer in unit of TCHAR. If either the size of read buffer or LFN working buffer is insufficient for the LFN or the object has no LFN, a null string will be returned to the LFN read buffer. If the LFN contains any character that cannot be converted to OEM code, a null string will be returned but this is not the case on Unicode API configuration. When lfname is NULL, nothing of the LFN is returned. When the object has no LFN, some small capitals can be contained in the SFN.

When relative path feature is enabled (_FS_RPATH == 1), "." and ".." entries are not filtered out and it will appear in the read entries.

This function is available if _FS_MINIMIZE <= 1.

Parameters

dj	Pointer to the open directory object
fno	Pointer to the file information to be set with next entry's information; fno == NUL means reset to first directory entry

Returns

FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_INVALID_OBJECT, FR_TIMEOUT, FR_NOT-_ENOUGH_CORE

2.55.4.13 FRESULT f_stat (const TCHAR ∗ *path*, FILINFO ∗ *fno*)

Get file status.

This function gets the size, timestamp and attribute of a file or directory. For details of the information, refer to the FILINFO structure and f_readdir().

This function is not available in minimization level of >= 1.

Parameters

path	Pointer to the directory's file name (0-terminated)
fno	Pointer to the file information

Returns

FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_NO_FILE, FR_NO_PATH, FR_INVALID_NA-ME, FR_INVALID_DRIVE, FR_NOT_ENABLED, FR_NO_FILESYSTEM, FR_TIMEOUT, FR_NOT_ENOUGH-_CORE

2.55.4.14 FRESULT f_write (FIL ∗ *fp*, const void ∗ *buff*, uint16_t *btw*, uint16_t ∗ *bw*)

Write data to a file.

The read/write pointer in the file object is increased by the number of bytes written. After the function succeeded, `*bw` should be checked to detect the disk full condition. In case of `*bw` < `btw` there was not enough space for the (complete) write operation.

The function may require more time when the volume is full or nearly so.

Parameters

fp	Pointer to the file object
buff	Pointer to data buffer
btw	Number of bytes to write This number must not be greater than the sector size (512)
bw	Pointer to result: number of bytes written

Returns

FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_INVALID_OBJECT, FR_TIMEOUT

2.55.4.15 FRESULT f_getfree (const uint8_t *vol*, uint32_t ∗ *nclst*)

Get the number of free clusters on the drive.

This function gets number of free clusters on the drive. The member `csize` in the file system object is the number of sectors per cluster, so that the free space in unit of sector can be calculated. When FSInfo structure on FAT32 volume is not in sync, this function may return an incorrect free cluster count.

This function is available if _FS_READONLY == 0 and _FS_MINIMIZE == 0.

Parameters

vol	the drive's volume number (0..9); to extract form a path name use extractDrive(&path)
nclst	Pointer to result: number of free clusters (not NULL!)

Returns

FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_INVALID_DRIVE, FR_NOT_ENABLED, FR_-NO_FILESYSTEM, FR_TIMEOUT

2.55.4.16 FRESULT f_truncate (FIL ∗ *fp*)

Truncate file.

This function truncates the file size to the current file read/write point. This function has no effect if the file read/write pointer is already pointing end of the file.

This function is available if _FS_READONLY == 0 and _FS_MINIMIZE == 0.

Parameters

fp	Pointer to the file object

Returns

FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_INVALID_OBJECT, FR_TIMEOUT

2.55.4.17 FRESULT f_sync (FIL ∗ *fp*)

Flush written / cached data to a file.

This function transfers write data caches in memory / sector buffers to the device. (The process and the related functions is normally called flush.) The function f_close() does, of course, the same before discarding the file object.

Calling this function periodically or after every f_write() can reduce the risk of data loss due to power failure or SMC removal at all costs of otherwise unnecessary device accesses.

Parameters

fp	Pointer to the file object

Returns

FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_INVALID_OBJECT, FR_TIMEOUT

2.55.4.18 FRESULT f_unlink (const TCHAR * *path*)

Delete an existing file or directory.

If the object to be erased / removed has a read-only attribute (AM_RDO) the operation will be rejected with FR_DENIED.

To remove a directory it must be empty and must not be current directory or the operation will be rejected with FR_DENIED.

A file to be removed must not be open or the FAT volume can collapse.

This function is available if _FS_READONLY == 0 and _FS_MINIMIZE == 0.

Parameters

path	Pointer to the directory's file name (0-terminated)

Returns

FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_NO_FILE, FR_NO_PATH, FR_INVALID_NAME, FR_DENIED, FR_EXIST, FR_WRITE_PROTECTED, FR_INVALID_DRIVE, FR_NOT_ENABLED, FR_NO_FILESYSTEM, FR_TIMEOUT, FR_LOCKED, FR_NOT_ENOUGH_CORE

2.55.4.19 FRESULT f_mkdir (const TCHAR * *path*)

Create a new directory.

This function is available if _FS_READONLY == 0 and _FS_MINIMIZE == 0.

Parameters

path	Pointer to the directory's file name (0-terminated)

Returns

FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_NO_PATH, FR_INVALID_NAME, FR_DENIED, FR_EXIST, FR_WRITE_PROTECTED, FR_INVALID_DRIVE, FR_NOT_ENABLED, FR_NO_FILESYSTEM, FR_TIMEOUT, FR_NOT_ENOUGH_CORE

2.55.4.20 FRESULT f_chmod (const TCHAR * *path*, uint8_t *value*, uint8_t *mask*)

Change attribute of the file or directory.

This function will set and/or clear the attributes AM_RDO (read only), AM_ARC (archive), AM_SYS (system) and AM_HID (hidden) of the file or directory specified. The parameters `value` and `mask` are OR expressions of the attribute masks named. Setting (`value`) has preference over clearing (`mask`).

This function is available if _FS_READONLY == 0 and _FS_MINIMIZE == 0.

Parameters

path	Pointer to the file name (0-terminated)
value	Attribute flags to be set
mask	Attribute flag to be cleared

Returns

FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_NO_FILE, FR_NO_PATH, FR_INVALID_NAME, FR_WRITE_PROTECTED, FR_INVALID_DRIVE, FR_NOT_ENABLED, FR_NO_FILESYSTEM, FR_TIMEOUT, FR_NOT_ENOUGH_CORE

2.55.4.21 FRESULT f_utime (const TCHAR ∗ *path,* const FILINFO ∗ *fno*)

Change time-stamps of a file or directory.

Parameters

path	Pointer to the file/directory name
fno	Pointer to a FILINFO structure with .fdate and .ftime values to be set to the file respectively directory

Returns

FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_NO_FILE, FR_NO_PATH, FR_INVALID_NAME, FR_WRITE_PROTECTED, FR_INVALID_DRIVE, FR_NOT_ENABLED, FR_NO_FILESYSTEM, FR_TIMEOUT, FR_NOT_ENOUGH_CORE

2.55.4.22 FRESULT f_rename (const TCHAR ∗ *path_old,* const TCHAR ∗ *path_new*)

Rename or move a file or directory.

This function renames an object (file or directory) and can also move it to other directory. The logical drive number is determined by old name, new name must not contain a logical drive number.

Do not rename open objects.

This function is available if _FS_READONLY == 0 and _FS_MINIMIZE == 0.

Parameters

path_old	Pointer to the current name / path
path_new	Pointer to the new name / path

Returns

FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_NO_FILE, FR_NO_PATH, FR_INVALID_NAME, FR_DENIED, FR_EXIST, FR_WRITE_PROTECTED, FR_INVALID_DRIVE, FR_NOT_ENABLED, FR_NO_FILESYSTEM, FR_TIMEOUT, FR_LOCKED, FR_NOT_ENOUGH_CORE

2.55.4.23 FRESULT f_fdisk (uint8_t *pdrv,* const uint32_t *szt[ ],* void ∗ *work*)

Divide a physical drive into some partitions.

This function creates a partition table into the MBR of the physical drive. The partitioning rule is in generic FDISK format so that it can create up to four primary partitions. Extended partition is not supported. The parameter `szt` specifies how to divide the physical drive. The first item specifies the size of first primary partition and fourth item specifies the fourth primary partition. If the value is less than or equal to 100, it means percentage of the partition in the entire disk space. If it is larger than 100, it means partition size in unit of sector.

This function is available if _MULTI_PARTITION == 2

Parameters

pdrv	Physical drive number
szt	Pointer to the size table for each partition
work	Pointer to the working buffer. The size must be at least _MAX_SS bytes.

Returns

FR_OK, FR_DISK_ERR, FR_NOT_READY, FR_WRITE_PROTECTED, FR_INVALID_PARAMETER

2.56 Configuration options for uIP

2.56.1 Overview

uIP is configured using the per-project configuration file uipopt.h. This file contains all compile-time options for uIP and should be tweaked to match each specific project. The uIP distribution contains a documented example "uipopt.h" that can be copied and modified for each project.

Note

Most of the configuration options in the uipopt.h should not be changed, but rather the per-project uip-conf.h file.

Files

- file uip-conf.h

  IP configuration file.
- file uipopt.h

  Configuration options for uIP.

Project-specific configuration options

uIP has a number of configuration options that can be overridden for each project.

These are kept in a project-specific uip-conf.h file and all configuration names have the prefix UIP_CONF.

- typedef uint8_t uip_tcp_appstate_t [SIZE_OF_BIGGEST_APPSTATE]

  The type of appstate in uip_conn.
- void uip_appcall (void)

  The uIP event function.
- void udp_appcall (void)

  The uIP udp event function.
- #define UIP_CONF_UDP_CONNS 10

  Maximum number of UDP connections.
- #define UIP_CONF_MAX_CONNECTIONS 10

  Maximum number of TCP connections.
- #define UIP_CONF_MAX_LISTENPORTS

  Maximum number of listening TCP ports.
- #define UIP_CONF_ARPTAB_SIZE 12

  The size of the ARP table.
- #define UIP_CONF_BUFFER_SIZE 620

  uIP buffer size
- #define UIP_CONF_BYTE_ORDER UIP_LITTLE_ENDIAN

  CPU byte order.
- #define UIP_CONF_LOGGING

  Logging on or off.
- #define UIP_CONF_UDP

  UDP support on or off.
- #define UIP_CONF_UDP_CHECKSUMS 1

  UDP checksums on or off.
- #define UIP_CONF_INC_CHECKSUMS 0

  Check incoming checksums on or off.

- #define UIP_CONF_STATISTICS 0

  uIP statistics on or off
- #define **UIP_CONF_BROADCAST**
- #define UIP_APPCALL

  Macro to name the uIP event function.
- #define UIP_UDP_APPCALL

  Macro to name the uIP udp event function.

Static configuration options

These configuration options can be used for setting the IP address settings statically, but only if UIP_FIXEDADDR is set to 1.

The configuration options for a specific node includes IP address, netmask and default router as well as the Ethernet address. The netmask, default router and Ethernet address are applicable only if uIP should be run over Ethernet.

All of these should be changed to suit your project.

- #define UIP_FIXEDADDR

  Determines if uIP should use a fixed IP address or not.
- #define UIP_PINGADDRCONF

  Ping IP address assignment.

IP configuration options

- #define UIP_TTL

  The IP TTL (time to live) of IP packets sent by uIP.
- #define UIP_REASSEMBLY

  Turn on support for IP packet re-assembly.
- #define UIP_REASS_MAXAGE

  maximum wait time an IP fragment in the re-assembly buffer

UDP configuration options

- #define UIP_UDP

  UDP support should be compiled in (or not)
- #define UIP_UDP_CHECKSUMS

  Checksums should be used (or not)
- #define UIP_INC_CHECKSUMS 0

  Check incoming checksums on or off.
- #define UIP_UDP_CONNS

  Maximum amount of concurrent UDP connections.

TCP configuration options

- #define UIP_ACTIVE_OPEN

  Support for opening connections from uIP should be compiled in (ot not)
- #define UIP_CONNS

  The maximum number of simultaneously open TCP connections.
- #define UIP_LISTENPORTS

  The maximum number of simultaneously listening TCP ports.

- #define UIP_URGDATA

  Support for TCP urgent data notification should be compiled in (or not)
- #define UIP_RTO

  The initial retransmission timeout counted in timer pulses.
- #define UIP_MAXRTX

  The maximum number of times a segment should be retransmitted.
- #define UIP_MAXSYNRTX

  The maximum number of times a SYN segment should be retransmitted.
- #define UIP_TCP_MSS

  The TCP maximum segment size.
- #define UIP_RECEIVE_WINDOW

  The size of the advertised receiver's window.
- #define UIP_TIME_WAIT_TIMEOUT

  How long a connection should stay in the TIME_WAIT state.

ARP configuration options

- #define UIP_ARPTAB_SIZE

  The size of the ARP table.
- #define UIP_ARP_MAXAGE

  The maximum age of ARP table entries measured in 10 seconds unit.

General configuration options

- void uip_log (char ∗msg)

  Print out a uIP log message.
- #define UIP_BUFSIZE

  The size of the uIP packet buffer.
- #define UIP_STATISTICS

  Statistics support should be compiled in (or not)
- #define UIP_LOGGING

  Logging of certain events should be compiled in (or not)
- #define UIP_BROADCAST

  Broadcast support.
- #define UIP_LLH_LEN

  The link level header length.

CPU architecture configuration

The CPU architecture configuration is where the endianess of the CPU on which uIP is to be run is specified.

Most CPUs today are little endian, including Intel (x86) and ATmega. The most notable exception are Motorola's CPUs which are big endian. The BYTE_ORDER macro must reflect the CPU architecture on which uIP is to be run.

- #define UIP_BYTE_ORDER

  The byte order of the CPU on which uIP is to be run.
- #define UIP_ARCH_ADD32

  There is a platform implementation of 32bit big endian addition.

Application specific configurations

An uIP application is implemented using a single application function that is called by uIP whenever a TCP/IP event occurs.

The name of this function must be registered with uIP at compile time using the UIP_APPCALL definition.

uIP applications can store the application state within the uip_conn structure by specifying the type of the application structure by typedef'ing the type uip_tcp_appstate_t and uip_udp_appstate_t.

The file containing the definitions must be included in the uipopt.h file.

The following example illustrates how this can look.

```
void httpd_appcall(void);
#define UIP_APPCALL     httpd_appcall

struct httpd_state {
  uint8_t state;
  uint16_t count;
  char *dataptr;
  char *script;
};
typedef struct httpd_state uip_tcp_appstate_t
```

- typedef uint16_t uip_udp_appstate_t

  The type of appstate in an uip_udp_conn.

2.56.2 Define Documentation

2.56.2.1 #define UIP_CONF_UDP_CONNS 10

Maximum number of UDP connections.

This value determines the length of the UDP connections list (array uip_udp_conns). As the uip_udp_conn structure is quite small ($\sim$ 11 byte) this value is not critical.

See also: UIP_UDP_CONNS

2.56.2.2 #define UIP_CONF_MAX_CONNECTIONS 10

Maximum number of TCP connections.

This value determines the length of the TCP connections list (array uip_conns). As the uip_conn structure is very large ($>$ kByte if http is used) this value is critical.

See also: UIP_CONNS

2.56.2.3 #define UIP_CONF_MAX_LISTENPORTS

Maximum number of listening TCP ports.

See also: UIP_LISTENPORTS

2.56.2.4 #define UIP_CONF_ARPTAB_SIZE 12

The size of the ARP table.

This option should be set to a larger value if this uIP node will have many connections from the local network. (uIP default would be 8)

2.56.2.5 #define UIP_CONF_BUFFER_SIZE 620

uIP buffer size

This is the size of uIP's global transfer (RAM) buffer uIP uses for sending and receiving, also for prepended ARP.

A.D. used 420 byte. A value > 590 allows to work with some DHCP servers (like Fritz!boxes) without restrictions.

2.56.2.6 #define UIP_CONF_BYTE_ORDER UIP_LITTLE_ENDIAN

CPU byte order.

ATmega and AVR GCC used in weAutSys are little endian.

With this setting for uIP we are on the save side. uIP does have this macro to be architecture independent, but there is uIP code just assuming little endian.

2.56.2.7 #define UIP_CONF_LOGGING

Logging on or off.

Used to set UIP_LOGGING

2.56.2.8 #define UIP_CONF_UDP

UDP support on or off.

Used to set UIP_UDP

2.56.2.9 #define UIP_CONF_UDP_CHECKSUMS 1

UDP checksums on or off.

Used to set UIP_UDP_CHECKSUMS

2.56.2.10 #define UIP_CONF_INC_CHECKSUMS 0

Check incoming checksums on or off.

Consideration and experiment show this can safely be turned off. (ENC checks incoming aggregate checksum and drops spoiled packets.

Used to set UIP_INC_CHECKSUMS

2.56.2.11 #define UIP_CONF_STATISTICS 0

uIP statistics on or off

Used to set UIP_STATISTICS

2.56.2.12 #define UIP_APPCALL

Macro to name the uIP event function.

An uIP application is implemented using a single application function that is called by uIP whenever a TCP/IP event occurs. The name of this function must be registered with uIP at compile time using this UIP_APPCALL definition.

The function named must return void and take no arguments. It will be called on uIP events.

The applications provided by uIP (like Telnet etc.) will set this macro to their specific event function, if not yet set. If there are multiple applications the function set here must handle them all, e.g. by acting as forwarder to the application specific functions.

uIP applications can store the application state within the uip_conn structure by specifying the type of the application structure by typedef'ing the type uip_tcp_appstate_t and uip_udp_appstate_t.

Remark by A.W.: A.D.'s typedef'ing approach (recommended here some ten years ago) does not carry further than to single TCP/IP application. But using more than one application and hence contradictory definitions (distributed all over original uIP's sources) causes trouble. Obviously (by now) a union of all state structures by all applications used is what A.D. must have meant in the end and what is done by weAutSys.

See also: uip_appcall

2.56.2.13 #define UIP_UDP_APPCALL

Macro to name the uIP udp event function.

See also

handle_dhcp()
udp_appcall
UIP_APPCALL

2.56.2.14 #define UIP_FIXEDADDR

Determines if uIP should use a fixed IP address or not.

If uIP should use a fixed IP address, the settings are set in the uipopt.h file. If not, the macros uip_sethostaddr(), uip_setdraddr() and uip_setnetmask() should be used instead.

2.56.2.15 #define UIP_PINGADDRCONF

Ping IP address assignment.

uIP uses a "ping" packets for setting its own IP address if this option is set. If so, uIP will start with an empty IP address and the destination IP address of the first incoming "ping" (ICMP echo) packet will be used for setting the hosts IP address.

Note

This works only if UIP_FIXEDADDR is 0.

2.56.2.16 #define UIP_TTL

The IP TTL (time to live) of IP packets sent by uIP.

This should normally not be changed.

2.56.2.17 #define UIP_REASSEMBLY

Turn on support for IP packet re-assembly.

uIP supports re-assembly of fragmented IP packets. This feature requires an additional amount of RAM to hold the re-assembly buffer and the re-assembly code size is approximately 700 bytes. The reassembly buffer is of the same size as the uip_buf buffer (configured by UIP_BUFSIZE).

Note

IP packet re-assembly is not heavily tested.

2.56.2.18 #define UIP_REASS_MAXAGE

maximum wait time an IP fragment in the re-assembly buffer

If the specified time expires, the fragment will be dropped.

2.56.2.19 #define UIP_UDP_CHECKSUMS

Checksums should be used (or not)

Note

Support for UDP checksums is currently not included in uIP, so this option has no function [AD] (which is not quite true [AW]).

2.56.2.20 #define UIP_INC_CHECKSUMS 0

Check incoming checksums on or off.

Consideration and experiment show this can safely be turned off. (ENC checks incoming aggregate checksum and drops spoiled packets.

Used to set UIP_INC_CHECKSUMS

2.56.2.21 #define UIP_ACTIVE_OPEN

Support for opening connections from uIP should be compiled in (ot not)

If the applications that are running on top of uIP for this project do not need to open outgoing TCP connections, this configuration option can be turned off to reduce the code size of uIP.

2.56.2.22 #define UIP_CONNS

The maximum number of simultaneously open TCP connections.

Since the TCP connections are statically allocated, turning this configuration knob down results in less RAM used. Each TCP connection requires approximately 30 bytes of memory.

2.56.2.23 #define UIP_LISTENPORTS

The maximum number of simultaneously listening TCP ports.

Each listening TCP port requires 2 bytes of memory.

2.56.2.24 #define UIP_URGDATA

Support for TCP urgent data notification should be compiled in (or not)

Urgent data (out-of-band data) is a rarely used TCP feature that would be required quite seldom.

2.56.2.25 #define **UIP_RTO**

The initial retransmission timeout counted in timer pulses.

This should not be changed.

2.56.2.26 #define **UIP_MAXRTX**

The maximum number of times a segment should be retransmitted.

Thereafter the connection should be aborted.

This should not be changed.

2.56.2.27 #define **UIP_MAXSYNRTX**

The maximum number of times a SYN segment should be retransmitted.

Thereafter the a connection request should be deemed to have been unsuccessful.

This should not need to be changed.

2.56.2.28 #define **UIP_TCP_MSS**

The TCP maximum segment size.

This is should not be to set to more than UIP_BUFSIZE - UIP_LLH_LEN - UIP_TCPIP_HLEN.

2.56.2.29 #define **UIP_RECEIVE_WINDOW**

The size of the advertised receiver's window.

Should be set low (i.e., to the size of the uip_buf buffer) is the application is slow to process incoming data, or high (32768 bytes) if the application processes data quickly.

2.56.2.30 #define **UIP_TIME_WAIT_TIMEOUT**

How long a connection should stay in the TIME_WAIT state.

This configuration option has no real implication, and it should be left untouched.

2.56.2.31 #define **UIP_ARPTAB_SIZE**

The size of the ARP table.

This option should be set to a larger value if this uIP node will have many connections from the local network.

2.56.2.32 #define **UIP_ARP_MAXAGE**

The maximum age of ARP table entries measured in 10 seconds unit.

A value 120 corresponds to 20 minutes life time of an ARP entry (the BSD default). As the ARP tick and the ARP entry stamps are unsigned 8 bit the value must be < 255.

In a stable (automation private) network environment invalidating ARP entries might be a nuisance as uIP might throw prepared send packets away if the ARP look-up fails — a burden to higher level protocols and the µController. It might be indicated to either raise this value or call uip_arp_timer less often in the runtime (weAutSys).

2.56.2.33 #define UIP_BUFSIZE

The size of the uIP packet buffer.

The uIP packet buffer should not be smaller than 60 bytes, and does not need to be larger than 1500 bytes. The lower the size the lower will be the TCP throughput.

See also: UIP_CONF_BUFFER_SIZE

2.56.2.34 #define UIP_STATISTICS

Statistics support should be compiled in (or not)

The statistics is useful for debugging and to show the user.

See also: UIP_CONF_STATISTICS

2.56.2.35 #define UIP_LOGGING

Logging of certain events should be compiled in (or not)

This is useful mostly for debugging. The function uip_log() must be implemented to suit the architecture of the project, if logging is turned on.

2.56.2.36 #define UIP_BROADCAST

Broadcast support.

This flag configures IP broadcast support. This is useful only together with UDP.

2.56.2.37 #define UIP_LLH_LEN

The link level header length.

This is the offset into the uip_buf where the IP header can be found. For Ethernet, this should be set to 14. For SLIP, this should be set to 0.

2.56.2.38 #define UIP_BYTE_ORDER

The byte order of the CPU on which uIP is to be run.

This option can be either BIG_ENDIAN (Motorola, SUN and some SIMATIC) or LITTLE_ENDIAN (Intel, ATmega and almost all else).

2.56.2.39 #define UIP_ARCH_ADD32

There is a platform implementation of 32bit big endian addition.

See also: add16littleTo32bigEndian

2.56.3 Typedef Documentation

2.56.3.1 typedef uint8_t uip_tcp_appstate_t[SIZE_OF_BIGGEST_APPSTATE]

The type of appstate in uip_conn.

uip_con.appstate will have to be cast to application specific types, usually struct struct thr_data_t.

2.56.3.2 typedef uint16_t **uip_udp_appstate_t**

The type of appstate in an uip_udp_conn.

The type of the application state that is to be stored in the uip_conn structure. This usually is typedef:ed to a struct holding application state information.

2.56.4 Function Documentation

2.56.4.1 void **uip_appcall** (void)

The uIP event function.

The reason for being called will be in `uip_flags`.

See also: UIP_APPCALL

2.56.4.2 void **udp_appcall** (void)

The uIP udp event function.

The reason for being called will be in the `uip_udp_conn` structure, especially in the ports.

See also: UIP_UDP_APPCALL

2.56.4.3 void **uip_log** (char * *msg*)

Print out a uIP log message.

This function must be implemented by the module that uses uIP, and is called by uIP whenever a log message is generated.

Chapter 3

Data Structure Documentation

3.1 appCLIreg_t Struct Reference

Data Fields

- p2ptFunA fun

  The user CLI thread's function (pointer to)
- char const ∗const ∗ userCommands

  Array of the (flash) user software command definitions.

3.1.1 Detailed Description

The user / application CLI registration type.

3.1.2 Field Documentation

3.1.2.1 p2ptFunA fun

The user CLI thread's function (pointer to)

The function pointed to must implement a user / application thread (behaviour) according to all Protothreads and weAutSys rules therefore.

This being NULL means there is no user command line interpreter (CLI).

3.1.2.2 char const∗ const∗ userCommands

Array of the (flash) user software command definitions.

This flash array of flash texts must be defined and initialised by user / application software. It may be initialised as NULL if the user software chooses to define / implement no own CLI commands.

Otherwise helpUserCm[] should be the first entry followed by (flash text) command definitions. The last entry must always be NULL like in the example:

```
char* userCommands[]) = { helpUserCm, // 0 no command just separator
    comADinputs, comADoff, // 1 2     + INDEX_OFFSET_LIST2
    comDiscoBun, comDemCount, // 3 4  + INDEX_OFFSET_LIST2
    NULL};
```

Index 0 (the first entry) will never be used or executed as command. It is to be used as the user commands headline for help (command list) output.

It is also essential that all command definitions made by user software are in flash memory (by INFLASH macro) the example:

```
INFLASH(char const comDiscoBun[]) = "discoBunny [t/10ms] Demo is disco lights
     \n";
INFLASH(char const comDemCount[]) = "countDemo  [t/10ms] Demo is DO count
     \n";
```

See also

INDEX_OFFSET_LIST2
FOLLOW_UP

The documentation for this struct was generated from the following file:

- include/we-aut_sys/cli.h

3.2 arp_entry Struct Reference

Data Fields

- struct uip_eth_addr ethaddr

    the 48 bit MAC address
- uint16_t ipaddr [2]

    the IP (V4) address
- uint8_t time

    Time stamp and valid flag.

3.2.1 Detailed Description

An ARP entry.

This type describes the result of one address resolution. It is a relation of a MAC and an IP address plus a time stamp of when this relation was entered first or updated.

3.2.2 Field Documentation

3.2.2.1 uint8_t time

Time stamp and valid flag.

The time stamp of this MAC/IP address relation is in terms of a ARP tick. The tick time (hence the unit) is usually 10 s. A longer value may be chosen to get longer ARP timeouts by an 8 bit tick and this 8 bit time stamp.

The value 0 is neither used as tick nor (hence) as time stamp value. `time` == 0 just means this entry is invalid.

The documentation for this struct was generated from the following file:

- include/uip/uip_arp.h

3.3 arp_hdr Struct Reference

Data Fields

- struct uip_eth_hdr ethhdr

the Ethernet header
- uint16_t hwtype

  hardware type (only Ethernet supported in uIP)

3.3.1 Detailed Description

The ARP header.

The documentation for this struct was generated from the following file:

- include/uip/uip_arp.h

3.4 cliThr_data_t Struct Reference

Data Fields

- uint8_t commandEnd

  end of command (=first token) if any
- uint8_t commandStart

  start of command (=first token) if any
- uint8_t commNumb

  The command number of first token.
- hierThr_data_t infoThread

  information threads and flags
- uint8_t length

  number of characters in line
- char line [LEN_OF_CLITHR_LINE+1]

  the input line (LEN_OF_CLITHR_LINE)
- uint8_t optionNumb

  The option number of first parameter or second token.
- outFlashTextThr_data_t outFlashTextThread

  for flash text output
- uint8_t paramEnd

  end of first parameter (=second / third token) if any
- uint8_t paramStart

  start of first parameter (=second / third token) if any
- pt_t pt

  The (raw) protothread data structure for the user CLI thread.
- FILE ∗ repStreams

  The streams used for output, report, reply.
- pt_t systCLIpt

  The (raw) protothread data structure for the system CLI thread.

3.4.1 Detailed Description

The organisational data for a command line interpreter (CLI) thread.

See also

registerAppCliThread
appCliThreadF
appCliThread
modThr_data_t

Examples:

main.c.

3.4.2 Field Documentation

3.4.2.1 uint8_t commNumb

The command number of first token.

If found by setCliLine() in the userCommands the value will be the index found there (starting at 0). If found in the systemCommands the value will be that found index + INDEX_OFFSET_LIST2.

255 means no matching token found. 254 says the command was ambiguously abbreviated. 0 is used as no command or nothing (else) to do. That schema is consistent with thr_data_t .flag usage.

Examples:

main.c.

3.4.2.2 uint8_t optionNumb

The option number of first parameter or second token.

If a second token was found by setCliLine() (and set in `paramStart`, `paramEnd`) and this parameter starts with a - (minus) followed by a non digit character it will be matched against a list of standard options. If a non ambiguous match is found `optionNumb` will be set to its number and `paramStart` will be set to the next parameter (third token) if found or to to 255. `paramEnd` will be set to the third token's end if given or stay at the option parameters end.

optNotGivenNum (255) means no matching option found. optAmbigousNum (254) says the option was ambiguously abbreviated. In those two cases `paramStart` and `paramEnd` still point to the (non) option parameter.

Examples:

main.c.

The documentation for this struct was generated from the following file:

- include/we-aut_sys/common.h

3.5 conf_data_t Struct Reference

Data Fields

- ipConf_t ipConf

  The IP configuration.
- eth_addr_t macAddress

  the MAC address
- uint8_t magicNumber

  The (magic) network configuration number.

3.5.1 Detailed Description

The device's basic configuration data type.

A structure of this type holds basic configuration data specific to the individual device. To survive power off one copy of these data is to be held in EEPROM at address eeConfigAdd = EEPROM[EEPROM_POINTER2_EE_CONF].

Examples:

individEEP.c.

3.5.2 Field Documentation

3.5.2.1 uint8_t **magicNumber**

The (magic) network configuration number.

This is a kind of major revision number for the configuration data structures and types. It must neither be 0 nor FF. It has to be changed whenever a software change would render previous EEPROM data invalid.

The value found in the EEPROM configuration data has to checked against that in defaultTypeConfData; this is done by persistInit().

The documentation for this struct was generated from the following file:

- include/we-aut_sys/persist.h

3.6 datdur_t Struct Reference

Data Fields

- uint16_t d

  The day.
- uint8_t h

  The hour in a day.
- uint8_t m

  The minute in an hour.
- uint8_t s

  The seconds in a minute.

3.6.1 Detailed Description

"Time since" as a structure: a duration or a date/time

This type stores the elapsed time as "seconds, .., days" relative to a arbitrary starting zero. It is the "structured" equivalent to a 32 bit unsigned "seconds since" value.

Depending on (the semantic of) this zero point in time this always represents a duration but in case of an appropriate "zero" also an absolute date and time.

Due to said equivalence an unsigned 32 bit seconds value can be converted to this type and vice versa. So it is commonly used as intermediate form for parsing and formatting.

3.6.2 Field Documentation

3.6.2.1 uint8_t s

The seconds in a minute.

This value will wrap from 59 to 0 (after one minute).

3.6.2.2 uint8_t m

The minute in an hour.

This value wrap from 59 to 0 (after one hour).

3.6.2.3 uint8_t h

The hour in a day.

This value wrap from 23 to 0 (after one day).

3.6.2.4 uint16_t d

The day.

This day counter will wrap after about 179 years.

If used as offset to 1st of March 2008 (1.3.2008 / 2008-03-01) this absolute date will get us to 2187 and hence far beyond 2100 respectively 2076 (2008 + 136/2), see also getDaysByDat().

The documentation for this struct was generated from the following file:

- include/we-aut_sys/timing.h

3.7 date_t Struct Reference

Data Fields

- uint8_t d

  The day in the month.
- uint8_t m

  The month.
- uint8_t wd

  The day in the week.
- uint8_t y

  The year.

3.7.1 Detailed Description

Date structure: a date in our time.

This type represents a date in the range 2000-01-01 to 2255-12-31 including day-of-week. Regarding the supplied functions for converting, formatting and parsing and its internal use it is strongly recommended to restrict its use to weAutSys's (currently) usable date range: 2008-3-1 .. +68 years.

In that restricted sense this is the "structured" equivalent to the absolute date as days since March 1st 2008 (weAutSys's day 0) — especially to the field d of a datdur_t structure related to that "zero".

Due to this equivalence an unsigned 16 bit "number of days since" value can be converted to this type and vice versa.

This type it is also commonly used as intermediate form for parsing and formatting.

3.7.2 Field Documentation

3.7.2.1 uint8_t wd

The day in the week.

The range is 1..7 Sunday to Saturday.

The value 0 means unknown respectively not yet set consistent to the other fields.

3.7.2.2 uint8_t y

The year.

The (recommended / usable) range is 8..141 to be interpreted as 2008 .. 2141

3.7.2.3 uint8_t m

The month.

The range is, of course, 1..12 as January..December.

3.7.2.4 uint8_t d

The day in the month.

The range is 1..28 | 29 | 30 | 31.

The documentation for this struct was generated from the following file:

- include/we-aut_sys/timing.h

3.8 dhcpMsg_t Struct Reference

Data Fields

- uint8_t xid [4]

    To match incoming DHCP messages with pending requests.

3.8.1 Detailed Description

The DHCP message structure.

The documentation for this struct was generated from the following file:

- include/uip/apps/dhcpc.h

3.9 dhcpOption_t Struct Reference

Data Fields

- uint8_t code

  the option code
- uint8_t length

  the length of value (0 implied if code == DHCP_OPTION_END)
- char value [ ]

  the value(s); an array of length bytes

3.9.1 Detailed Description

The structure of a DHCP message option field.

The documentation for this struct was generated from the following file:

- include/uip/apps/dhcpc.h

3.10 dhcpState_t Struct Reference

Data Fields

- struct uip_udp_conn ∗ conn

  pointer to the connection (structure)
- uint16_t ipaddr [2]

  the client's wanted / granted IP address
- const void ∗ mac_addr

  pointer to device's MAC address
- struct pt pt

  the protothreads (raw) structure
- uint8_t serverid [4]

  the DHCP server's IP address
- char state

  DHCP state machine state.
- struct timer_t timer

  timer (in ms while in protocol; in seconds renew time)

3.10.1 Detailed Description

Structure for DHCP application state.

The documentation for this struct was generated from the following file:

- include/uip/apps/dhcpc.h

3.11 DIR Struct Reference

Data Fields

- uint32_t clust
  Current cluster.
- uint8_t ∗ dir
  Pointer to the current SFN entry in the win[].
- uint8_t ∗ fn
  Pointer to the SFN (in/out) {file[8],ext[3],status[1]}.
- FATFS ∗ fs
  Pointer to the owner file system object.
- uint16_t id
  Owner file system mount ID.
- uint16_t index
  Current read/write index number.
- uint32_t sclust
  Table start cluster (0: Root dir)
- uint32_t sect
  Current sector.

3.11.1 Detailed Description

Directory object structure (DIR)

A structure of this type holds the state of an open respectively selected directory initialised by f_opendir and subsequently used by directory functions.

See also

FIL

The documentation for this struct was generated from the following file:

- include/fatFS/ff.h

3.12 dst_rule_year_t Struct Reference

Data Fields

- uint8_t firstSunNovember
  Day in month of first Sunday in November.
- uint32_t firstSunNovember0200offset
  Number of seconds from March 1st to first Sunday in November 02:00.
- uint8_t lastSunMarch
  Day in month of last Sunday in March.
- uint32_t lastSunMarch0100offset
  Number of seconds from March 1st to last Sunday in March 01:00.
- uint8_t lastSunOctober
  Day in month of last Sunday in October.
- uint32_t lastSunOctober0100offset
  Number of seconds from March 1st to last Sunday in October 01:00.

- uint8_t secondSunMarch

  Day in month of second Sunday in March.
- uint32_t secondSunMarch0200offset

  Number of seconds from March 1st to second Sunday in March 02:00.

3.12.1 Detailed Description

Rule structure: DST rules for a given (set of) year(s)

This type represents the daylight-saving time rules by a set of values needed by the the DST handling functions. The rules represented by a structure of this type support the EU and US DST rules.

The current (as of 2012) EU and US rules are made available for the years 2000 .. 2255. This will be longer than weAutSys may be used and especially longer than politicians will stick to the current EU and US rules.

Note: The statements (last "Sunday in ..") of the rules supplied will, of course, hold — but the DST laws will probably change.

As all dates and offsets for US and EU rules given here are after March 1st or are relative to this date the values will not be affected by leap day (February 29th). Hence (non regarding the infernal leap seconds) there are just 7 different sets of rules resp. rule structures numbered 0 .. 6. This rule number may be obtained by lastSunMarch - 25 i.e. (first byte of structure) - 25.

See also

getDSTrule()

3.12.2 Field Documentation

3.12.2.1 uint8_t lastSunMarch

Day in month of last Sunday in March.

For the set of years for which this rule is valid this will give the last Sunday in March. This is when EU switches DST on at 01:00 UTC, by turning the local times 1 hour ahead.

range: 25 .. 31

3.12.2.2 uint8_t lastSunOctober

Day in month of last Sunday in October.

For the set of years for which this rule is valid this will give the last Sunday in October. This is when EU switches DST off at 01:00 UTC, by turning the local times 1 hour back to normal. This will repeat the local time 02:00,000 .. 02:59,999 in CET, for example.

range: 25 .. 31

3.12.2.3 uint32_t lastSunMarch0100offset

Number of seconds from March 1st to last Sunday in March 01:00.

This is the number of seconds from March 1st 00:00 to last Sunday in March 01:00. This value supports the exact timing of EU switching DST on.

3.12.2.4 uint32_t lastSunOctober0100offset

Number of seconds from March 1st to last Sunday in October 01:00.

This is the number of seconds from March 1st 00:00 to last Sunday in October 01:00. This value supports the exact timing of EU switching DST off.

3.12.2.5 uint8_t secondSunMarch

Day in month of second Sunday in March.

For the set of years for which this rule is valid this will give the second Sunday in March. This is when most of the United States of America, Canada and Mexico switch DST on at 02:00 (AM) local timeTC, by turning the local times 1 hour ahead.

range: 8 .. 14

3.12.2.6 uint8_t firstSunNovember

Day in month of first Sunday in November.

For the set of years for which this rule is valid this will give the first Sunday in November. This is when most Northern America witches DST off at 02:00 (AM).

range: 1..7

3.12.2.7 uint32_t secondSunMarch0200offset

Number of seconds from March 1st to second Sunday in March 02:00.

This is the number of seconds from March 1st 00:00 to second Sunday in March 02:00. This value supports the exact timing of US etc. switching DST on.

3.12.2.8 uint32_t firstSunNovember0200offset

Number of seconds from March 1st to first Sunday in November 02:00.

This is the number of seconds from March 1st 00:00 to first Sunday in November 02:00. This value supports the exact timing of US etc. switching DST off.

The documentation for this struct was generated from the following file:

- include/we-aut_sys/timing.h

3.13 eeOp_data_t Struct Reference

Data Fields

- uint16_t eeAddress

  The EEPROM address to start the operation.
- uint8_t noOfWrite

  Number of bytes to write.
- pt_t pt

  The (raw) protothread data structure.
- uint8_t ∗ writeBuffer

  Pointer to the buffer of data to be written.

3.13.1 Detailed Description

The data structure for an EEPROM bulk operation thread.

A structure of this type holds all data for an EEPROM write operation on 1..255 continuous bytes. It will be passed as parameter to the thread function eeOperationThread.

See also

initSetAppThread(struct mThr_data_t ∗, p2ptFun)

3.13.2 Field Documentation

3.13.2.1 uint16_t eeAddress

The EEPROM address to start the operation.

It has to be set to the first destination address in EEPROM before the thread function is called the first time.

It will be incremented after each step.

3.13.2.2 uint8_t∗ writeBuffer

Pointer to the buffer of data to be written.

The new data for the EEPROPM are taken from here.

This pointer will be incremented after every step.

For the thread function eeOperationThread this is an address in RAM.

3.13.2.3 uint8_t noOfWrite

Number of bytes to write.

It has to be set accordingly before the thread function is called the first time. This value will be decremented after every step.

0, of course, means no (further) operations pending.

The documentation for this struct was generated from the following file:

- include/we-aut_sys/persist.h

3.14 ethip_hdr Struct Reference

Data Fields

- struct uip_eth_hdr ethhdr
  the Ethernet header

3.14.1 Detailed Description

The IP header.

The documentation for this struct was generated from the following file:

- include/uip/uip_arp.h

3.15 FATFS Struct Reference

Data Fields

- uint8_t csize

  Sectors per cluster (1,2,4...128)
- uint32_t database

  Data start sector.
- uint32_t dirbase

  Root directory start sector (FAT32: Cluster#)
- uint8_t drv

  Physical drive number.
- uint32_t fatbase

  FAT start sector.
- uint32_t free_clust

  Number of free clusters.
- uint8_t fs_type

  FAT sub-type (0:Not mounted)
- uint8_t fsi_flag

  fsinfo dirty flag (1:must be written back)
- uint32_t fsi_sector

  fsinfo sector (FAT32)
- uint32_t fsize

  Sectors per FAT.
- uint16_t id

  File system mount ID.
- uint32_t last_clust

  Last allocated cluster.
- uint32_t n_fatent

  Number of FAT entries (= number of clusters + 2)
- uint8_t n_fats

  Number of FAT copies (1,2)
- uint16_t n_rootdir

  Number of root directory entries (FAT12/16)
- uint8_t wflag

  win[] dirty flag (1:must be written back)
- uint8_t win [_MAX_SS]

  Disk access window for directory, FAT and data.
- uint32_t winsect

  Current sector appearing in the win[].

3.15.1 Detailed Description

File system object structure (FATFS)

3.15.2 Field Documentation

3.15.2.1 uint8_t win[_MAX_SS]

Disk access window for directory, FAT and data.

This window respectively buffer is bound to the file system structure (FATFS) on (fiexd) tiny configuration.

Note (internal, future): On weAutSys' SMC this should be omitted as the (driver's) sector buffer is of same size and purpose. It also handles "dirty state" correctly / automatically.

The documentation for this struct was generated from the following file:

- include/fatFS/ff.h

3.16 FIL Struct Reference

Data Fields

- uint32_t clust

  Current cluster of fpter.
- uint8_t ∗ dir_ptr

  Pointer to the directory entry in the window.
- uint32_t dir_sect

  Sector containing the directory entry.
- uint32_t dsect

  Current data sector of fpter.
- uint8_t flag

  File status flags.
- uint32_t fptr

  File read/write pointer (0ed on file open)
- FATFS ∗ fs

  Pointer to the related file system object.
- uint32_t fsize

  File size.
- uint16_t id

  File system mount ID of the related file system object.
- uint8_t pad1

  Fill byte.
- uint32_t sclust

  File data start cluster (0:no data cluster, always 0 when fsize is 0)

3.16.1 Detailed Description

File object structure (FIL)

A structure of this type holds the state of an open file initialised by f_open() and subsequently used by file functions.

Internal hint: some parts of ChaN's fatFS software assume the beginning of DIR and FIL being identical.

See also

DIR

The documentation for this struct was generated from the following file:

- include/fatFS/ff.h

3.17 FILINFO Struct Reference

Data Fields

- uint8_t fattrib

    Attribute See AM_RDO, AM_DIR.
- uint16_t fdate

    Last modified date.
- TCHAR fname [13]

    Short file name (8.3 format, 0-terminated)
- uint32_t fsize

    File size.
- uint16_t ftime

    Last modified time.
- uint8_t namLen

    Length of file name; 0..12 for short names.

3.17.1 Detailed Description

File status structure (FILINFO)

A structure of this type holds some extra informations on a file or directory. It is determined by f_stat() for later reference.

The documentation for this struct was generated from the following file:

- include/fatFS/ff.h

3.18 FS_WORK Struct Reference

Data Fields

- DIR dir

    Current directory's status structure.
- FIL fil

    Current file's status structure.
- FILINFO filInf

    Current file's or directory's information structure.
- FRESULT fRes

    Result of file system operations.
- FRESULT fRes1

    Result of file system operations.
- FRESULT fRes2

    Result of file system operations.
- uint8_t lock

    Lock state of the file operation structure.
- char ∗ path

    A current file's or directory's name.
- ucnt32_t szi32

    Current (32 bit) result or parameter value.

3.18.1 Detailed Description

Work space for file system operations (structure FS_WORK)

A structure of this type holds the state of the system or application software's ongoing file system operations.

See also

fsWork

3.18.2 Field Documentation

3.18.2.1 uint8_t lock

Lock state of the file operation structure.

A value 0 means that this structure is not locked or the implementation chooses not to use / implement that feature.

The (reserved) value SMC_FS_SYSTUSE says that the weAutSys runtime itself is in the process of file system operations and uses this structure for state. This will only happen by respective (human) commands.

See also

SMC_FS_CLIUSE
SMC_FS_SYSTUSE
lockFsWorkFor
unlockFsWorkFrom

3.18.2.2 char∗ path

A current file's or directory's name.

This is a pointer to a null terminated string holding a path. Its content is read only for all file system functions.

3.18.2.3 FRESULT fRes

Result of file system operations.

This variable is to hold the last main file system operation result.

Examples:

main.c.

3.18.2.4 FRESULT fRes1

Result of file system operations.

This variable is to hold the last secondary file system operation result.

Examples:

main.c.

3.18.2.5 FRESULT fRes2

Result of file system operations.

This variable is to hold the last main tertiary file system operation result.

Examples:

main.c.

3.18.2.6 FILINFO filInf

Current file's or directory's information structure.

This variable is to hold one file's state (over thread yields).

3.18.2.7 DIR dir

Current directory's status structure.

This variable is to hold one directory's state (over thread yields).

3.18.2.8 FIL fil

Current file's status structure.

This variable is to hold one file's state (over thread yields).

Examples:

main.c.

3.18.2.9 ucnt32_t szi32

Current (32 bit) result or parameter value.

The documentation for this struct was generated from the following file:

- include/we-aut_sys/smc2fs.h

3.19 hierThr_data_t Struct Reference

Data Fields

- uint8_t flag

  primary flag
- uint8_t flag2

  secondary flag
- uint8_t flag3

  extra flag
- uint8_t flag4

  extra secondary flag
- pt_t pt

  the (raw) protothread data structure
- pt_t pt2

  the (raw) protothread data structure for the secondary or sub thread

3.19.1 Detailed Description

The organisational data for a small thread hierarchy.

This structure defines two protothreads (raw) structures and two flags. The intended use is for one main threads that can have one ore more sub-threads.

See also

outFlashTextThreadF
initOutFlashTextThread

The documentation for this struct was generated from the following file:

- include/we-aut_sys/common.h

3.20 ipConf_t Struct Reference

Data Fields

- uip_ipaddr_t defaultRouter

  The default router.
- uip_ipaddr_t dns2Addr

  The name server 2.
- uip_ipaddr_t dnsAddr

  The name server.
- uip_ipaddr_t ipAddr

  The IP address.
- uint32_t leaseTime

  Lease time in seconds.
- uip_ipaddr_t netMask

  The netmask.
- uip_ipaddr_t ntp2Addr

  The time server 2.
- uip_ipaddr_t ntpAddr

  The time server.
- uint8_t setFlags

  Flags for the field is valid / is set.
- uint32_t setTime

  Set time.

3.20.1 Detailed Description

The IP configuration.

3.20.2 Field Documentation

3.20.2.1 uip_ipaddr_t dnsAddr

The name server.

This is the IP address of the name server 1.

weAutSys and ref uip "uIP" dispense with having multiple DNS servers.

3.20.2.2 uip_ipaddr_t dns2Addr

The name server 2.

This is the IP address of the name server 2.

If just one DNS server is available / set this should be dnsAddr.

weAutSys and ref uip "uIP" dispense with having multiple DNS servers.

3.20.2.3 uip_ipaddr_t ntpAddr

The time server.

This is the IP address of the NTP server 1 (if set).

3.20.2.4 uip_ipaddr_t ntp2Addr

The time server 2.

This is the IP address of the NTP server 2.

If just one NTP server is available / set this should be ntpAddr.

3.20.2.5 uint32_t leaseTime

Lease time in seconds.

If bit 1 (DHCP_MSK) is set in setFlags (part of) this configuration was set by a DHCP server and this is the real lease time in seconds (normal byte order).

Otherwise this element might be used as the requested lease time.

3.20.2.6 uint32_t setTime

Set time.

This is the run time (with seconds resolution) when this configuration was set. If bit 1 (DHCP_MSK) is set this is the time of the DHCP server's positive answer.

Rationale to use the run time instead of an absolute local time is a) the first one having no settings and gaps giving allways a correct "age" of the setting and b) the second one usually being set after the just set DHCP named one or two NTP servers and one of them gave the first correct absolute time.

3.20.2.7 uint8_t setFlags

Flags for the field is valid / is set.

Bit 0 : This configuration (or part of it) was set by responses from a DHCP server (DHCP_MSK).

Bit 4 : name server 1 is set (DNS1_MSK)

Bit 5 : name server 2 is set (DNS2_MSK)

Bit 6 : time server 1 is set (NTP1_MSK)

Bit 7 : time server 2 is set (NTP2_MSK)

The documentation for this struct was generated from the following file:

- include/we-aut_sys/network.h

3.21 modConfData_t Struct Reference

Data Fields

- p2ptFunM appModFun

  The application specific Modbus function (pointer to)
- uint8_t options

  Configuration options of the Modbus server.
- uint16_t port

  The port of the Modbus server.
- uint8_t unit

  The unit number of the Modbus server.

3.21.1 Detailed Description

The configuration data (type) for a Modbus handling.

3.21.2 Field Documentation

3.21.2.1 p2ptFunM appModFun

The application specific Modbus function (pointer to)

The function registered here must implement the user / application specific Modbus handler as thread.

A NULL value here means that there is no user / application specific Modbus handler and Modbus server functions (if any) are restricted to those defined by the weAutSys runtime.

To do any process I/O or handling through Modbus user software must provide and register an appropriate function. This user function is relieved from (almost) all communication and protocol handling: it must just consume and/or deliver the respective data. (Note that 16 bit units are in wrong endianes due to Modbus standard.)

As provided as protothread function the user software handling of a Modbus request may block or yield. In that case the next schedule would be at the next polling the open connection. That could be 100..1000ms later. This may be appropriate in special cases and with a good explanatory statement. The recommendation is to handle the request in one stint. Anyway, the data delivered by the request have to be consumed (copied) completely in the first and the data to be send have to be given completely in the last stint.

3.21.2.2 uint16_t port

The port of the Modbus server.

This is and should be the well known Modbus port. This is only provided for future use, should it be decided to make the Modbus server port configurable.

3.21.2.3 uint8_t unit

The unit number of the Modbus server.

If this server is to check the unit number in requests this is the only one accepted. Other wise the request's unit number and this value will be ignored (i.e. all will be accepted).

3.21.2.4 uint8_t options

Configuration options of the Modbus server.

Bit 0 (0x01) : if set check request's unit number against .unit

Bit 1..7 : future use

The documentation for this struct was generated from the following file:

- include/we-aut_sys/common.h

3.22 modTelegr_t Struct Reference

Data Fields

- ucnt16_t byToFol

  Bytes in the telegram to followProtocol identifier.
- uint8_t fCode

  The code of the function to perform.
- ucnt16_t pduCont [6]

  The (start of the) PDU content.
- ucnt16_t protId

  Protocol identifier.
- ucnt16_t transId

  Transaction identifier.
- uint8_t unitNumb

  The unit number.

3.22.1 Detailed Description

The (start of a) Modbus TCP/IP telegram.

The first seven bytes are the so called MBAB header.

The next 13 bytes are the (start / part of) the the so called PDU, the first byte of which always is the function code (1..127 / 0x01 0x7F) or error return code (129..255 / 0x81..0xFF).

See also

formModTelegr

3.22.2 Field Documentation

3.22.2.1 ucnt16_t transId

Transaction identifier.

Set by client — to be responded unchanged.

3.22.2.2 ucnt16_t protId

Protocol identifier.

Must be set 0x0000 by client (= Modbus) — and be so in response.

Remark: This word seems totally redundant in Modbus TCP/IP as the port (lport) identifies the protocol.

3.22.2.3 ucnt16_t byToFol

Bytes in the telegram to followProtocol identifier.

This is the byte count from and including the unit identifier byte. Is set by the client according to request length as well as by the server according to respose length.

Note 1: This as all 16 bit (word) values with Modbus is in wrong — big endian — byte order.

Note 2: This word is totally redundant in Modbus TCP/IP as it always must be telegram's length - 6.

3.22.2.4 uint8_t unitNumb

The unit number.

Is set by client and must responded unchanged.

An (user) implementation may decide to handle multiple units or to have an own special unit number and check if this number in the request is its own. In all else cases this value will be ignored.

3.22.2.5 uint8_t fCode

The code of the function to perform.

Is set by client request and usually copied as modThr_data_t.fcFlag for handler function / thread flag usage.

The value must responded unchanged in error free responses or with bit 7 (0x80) set as error response.

3.22.2.6 ucnt16_t pduCont[6]

The (start of the) PDU content.

If pairs of those 12 bytes are to be interpreted as 16 bit / word values they are in wronr byte order.

The documentation for this struct was generated from the following file:

- include/we-aut_sys/common.h

3.23 modThr_data_t Struct Reference

Data Fields

- struct modConfData_t config

  The system's Modbus configuration (copy of as of time of request)
- uint8_t dataModel

  Data model for in and / or out data.
- uint8_t errorCode

  Error code.
- uint8_t fcFlag

  The outstanding function to perform.
- uint16_t gotLen

  The complete length of the incoming Modbus request (telegram)
- uint8_t ∗ inData

  Incoming data (pointer to)
- uint8_t inDataByteCnt

  The length of the request's incoming data as byte count.
- ucnt16_t inDatTarg

The (logic) start or target address for the request's incoming data.
- uint8_t ∗ outData

    Outgoing data (pointer to)
- uint8_t outDataByteCnt

    The length of the responses outgoing data as byte count.
- ucnt16_t outDatSou

    The (logic) start or source address for the response's outgoing data.
- uint8_t pollCnt

    Poll count.
- pt_t pt

    The (raw) protothread data structure for the user modThr_data_t thread.
- struct modTelegr_t reqModTelegr

    A copy of (the start of the) incoming request.
- uint8_t retC

    The return code of the last schedule.
- ucnt32_t wrkV

    Working variables for the Modbus handling threads.

3.23.1 Detailed Description

The organisational data for a Modbus handler thread.

The similarity of structure of modThr_data_t data and those for cliThr_data_t CLI is not by chance.

Examples:

main.c.

3.23.2 Field Documentation

3.23.2.1 uint8_t fcFlag

The outstanding function to perform.

It is an accepted function code (FC) or 0. 0 is used as no code or nothing to do. That appointment is consistent with thr_data_t .flag usage.

3.23.2.2 uint8_t retC

The return code of the last schedule.

This is intended to hold the return code, that is the Modbus handler function's last return value.

3.23.2.3 struct modConfData_t config

The system's Modbus configuration (copy of as of time of request)

3.23.2.4 uint8_t pollCnt

Poll count.

Value 0 means first stint for handler function / tread; i.e. new data. Value 1 means second stint or first connection poll, and so on.

3.23.2.5 uint8_t dataModel

Data model for in and / or out data.

The values provided are Discrete inputs, Coil, Coils, Input registers, Mask single Holding and Holding registers. It will be set according to function code before the user / application Modbus handler is called.

Examples:

main.c.

3.23.2.6 uint8_t errorCode

Error code.

To be set (!=0) by the handler if the requested function cannot be performed.

Examples:

main.c.

3.23.2.7 ucnt16_t inDatTarg

The (logic) start or target address for the request's incoming data.

This value will be provided in correct byte order.

Examples:

main.c.

3.23.2.8 uint8_t* inData

Incoming data (pointer to)

This pointer is only valid if .inDatlen is not 0 and only in the first stint of a handling protothread function. For the type and interpretation of the data see primarily the function code.

Examples:

main.c.

3.23.2.9 uint8_t outDataByteCnt

The length of the responses outgoing data as byte count.

This field is set to expected value. It may be shortened by the handler thread / function if that decides to deliver less data that requested.

Examples:

main.c.

3.23.2.10 ucnt16_t outDatSou

The (logic) start or source address for the response's outgoing data.

This value will be provided in correct byte order.

Examples:

main.c.

3.23.2.11 uint8_t* outData

Outgoing data (pointer to)

This pointer is only valid if .outDataBytCnt is not 0 and only in the last stint of a handling protothread function.

Examples:

main.c.

3.23.2.12 ucnt32_t wrkV

Working variables for the Modbus handling threads.

This must be set according to the bytes copied to .outData (in the last stint) if any.

The documentation for this struct was generated from the following file:

- include/we-aut_sys/common.h

3.24 mThr_data_t Struct Reference

Data Fields

- uint8_t flag

  A flag byte for the threads specific usage.
- p2ptFun fun

  The thread's function (pointer to)
- pt_t pt

  The (raw) protothread data structure.
- uint8_t retC

  The return code of the last schedule.

3.24.1 Detailed Description

State data for a thread (minimal)

It is strongly recommended that threads use this structure for their essential state. The thread function's (the behaviour) recommended signature takes (only) a pointer to such structure as parameter.

Software using this schema is supported by some handling functions and macros provided. If more state info is required the extended type thr_data_t should be used; it can be cast back to this common smaller type mThr_data-_t to use the appropriate handling functions.

See also

resetAppThread
scheduleAppThread
scheduleRegdAppThread
scheduleFlgdAppThread
scheduleYldAppThread
registerAppThread(struct mThr_data_t ∗, p2ptFun)
initSetAppThread(struct mThr_data_t ∗, p2ptFun)

Examples:

main.c.

3.24.2 Field Documentation

3.24.2.1 uint8_t flag

A flag byte for the threads specific usage.

This is intended to hold (additionally to `pt`) the primary thread state. The semantic is defined by the thread function (the behaviour).

Insofar the thread implementation (i.e. the thread function code) is totally free with respect to this `flag's` semantic. The only fixed common convention is

0 : "no work to do" for the thread

This convention should be strictly observed, as weAutSys may not schedule a thread on `!flag`.

`flag` is used as workload count for cyclic threads respectively periodic tasks in weAutSys; that means something like

0 : "do no periodic work even if scheduled"

1 : "do normal periodic work"

2.. means cycle overrun "do the work of 2.. periods"

As said above, the value 0 meaning "nothing to do" is the only rule never to be violated.

Examples:

main.c.

3.24.2.2 uint8_t retC

The return code of the last schedule.

This is intended as a secondary status flag. The recommended (and supported) use is to hold the thread function's last return value.

Hint: Some supporting functions and macros use this `retC` to treat a yielding thread in a special way: it might be scheduled with flag == 0 for doing rest work but a blocking thread will not. Their criterion is just `retC ==` PT_YIELDED

3.24.2.3 p2ptFun fun

The thread's function (pointer to)

The function must implement a thread (behaviour) according to all Protothreads and weAutSys rules therefore. This pointer is binding this thread state to a thread function.

Hint / rationale: It is quite possible to have one thread function handle multiple thread objects implementing a common behaviour. This function must keep all thread state in the thread structure. But to have this state data never in in any other (static) variables is good practice anyway.

The other way round is never feasible: no living Protothread can be handled by more than one thread function at the same time.

Hence it is quite logical to (optionally) register this one / current thread function here.

This being NULL may hence mean this thread having no registered function respectively behaviour.

Hint 2: If the Protothread function is statically fixed for a set of threads (and none of the above mentioned scheduling macros is used on them) this `fun` pointer may be used for other purposes.

The documentation for this struct was generated from the following file:

- include/we-aut_sys/common.h

3.25 ntpMess_t Struct Reference

Data Fields

- uint8_t flags

  The flags.
- struct ntpTimestamp_t originateTimestamp

  The T1, see RFC 2030.
- uint8_t poll

  Maximum interval between successive messages.
- int8_t precision

  Servers time precision.
- struct ntpTimestamp_t receiveTimestamp

  The T2, see RFC 2030.
- ucnt32_t referenceIdentifier

  Identifies the particular reference clock.
- struct ntpTimestamp_t referenceTimestamp

  The timestamp of the last setting of the local clock.
- ucnt32_t rootDelay

  see RFC 2030
- ucnt32_t rootDispersion

  see RFC 2030
- uint8_t stratum

  A kind of time server hierarchy level.
- struct ntpTimestamp_t transmitTimestamp

  The T3, see RFC 2030.

3.25.1 Detailed Description

NTP message type.

See

Network Time Protocol Version 4 Protocol and Algorithms Specification [draft]

or see `RFC 4330`

Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI

and see also [RFC 2030, cites below]

NTP client operations [RFC 2030 quoted]:

```
      Timestamp name         ID   is set when
      ----------------------------------------------------------------------
      originateTimestamp     T1   request sent by client as transmitTimestamp
      receiveTimestamp       T2   request received by server
      transmitTimestamp      T3   reply sent by server
      destinationTimestamp   T4   reply received by client

   The roundtrip delay d and system clock offset t are defined as:

      d = (T4 - T1) - (T3 - T2)     t = ((T2 - T1) + (T3 - T4)) / 2
```

For the arithmetic operations on the timestamps use the functions ntpTimestampSum, ntpTimestampDif and ntp-TimestampHalf.

3.25.2 Field Documentation

3.25.2.1 uint8_t flags

The flags.

This byte consists of the leap indicator (2 Bits), a version number (3 bits) and mode (3 bits).

3.25.2.2 uint8_t poll

Maximum interval between successive messages.

The value is 2∗∗poll s.

Range: 4 .. 17 (16s .. 36h)

3.25.2.3 int8_t precision

Servers time precision.

The value is 2∗∗precision s.

Range: -6 .. -20 (16ms .. 1µs)

3.25.2.4 ucnt32_t referenceIdentifier

Identifies the particular reference clock.

Not used or evaluated here. See RFC 2030.

3.25.2.5 struct ntpTimestamp_t referenceTimestamp

The timestamp of the last setting of the local clock.

If the local clock has never been synchronized, the value is zero; see RFC 2030.

Not used or evaluated here.

The documentation for this struct was generated from the following file:

- include/we-aut_sys/ntp.h

3.26 ntpState_t Struct Reference

Data Fields

- uint8_t adjuState

  State / actions of actual and previous adjustment.
- struct uip_udp_conn ∗ conn

  pointer to the connection (structure)
- uint8_t count

  NTP state machine counter (used in some states)
- struct ntpTimestamp_t lastReceiveTimestamp

  our receive (T4)
- struct ntpTimestamp_t lastSendTimestamp

  our send / server' originate (T1)
- struct ntpTimestamp_t lastT21

  the last T2-T1
- struct ntpTimestamp_t lastT34

  the last T3-T4
- uint8_t logLev

  NTP repoprt level.
- uint16_t msDif

  Actual ms deviation.
- uint16_t msDifPrev

  Previous ms deviation.
- struct pt pt

  the protothreads (raw) structure
- uint32_t secDif

  Actual deviation in seconds.
- uint8_t state

  NTP state machine state.
- struct timer_t timer

  the protocol timer, seconds resolution, duration type

3.26.1 Detailed Description

Structure for NTP (client) application state.

3.26.2 Field Documentation

3.26.2.1 uint8_t state

NTP state machine state.

The lower NTP_STATE_SRVMSK two bits are 0 = no connection or the number (1, 2) of a configured NTP server connected to.

Bit 3 (8) means waiting for server's reply.

At the upper 4 bits 0x10 means first try respectively first request,

0xF0 says connection problematic or faulty,

0xB0 is server answer problematic (bogus, alarm),

0xE0 means connection established and

0xC0 signals continuous synchronisation.

3.26.2.2 uint8_t logLev

NTP repoprt level.

The higher the value the more events will be reported to the buffered log device.

0: uninitialised; will be set according to macro DEBUG_NTP + 3 when initialising NTP.

0..3: silent

4 : quiet (almost silent)

5 : info

6 : verbose

7 : debug

3.26.2.3 uint8_t adjuState

State / actions of actual and previous adjustment.

Bits 7,6 : 1 = last positive, 2 = last negative, 0 = last 0

Bits 5,4 : 0 = in the range -100 .. + 50ms

1 = in the range -999 .. +999 ms

2 = in the range +1s .. +1999 ms; 3 = outside -999 .. +1999 ms

Bits 4,5 : same as 6,7 for previous action

Bits 0,1 : same as 2,3 for previous action

3.26.2.4 uint32_t secDif

Actual deviation in seconds.

This is the current seconds deviation (mod 2∗∗32 in normal byte order). The effective sign of the correction is Bit 7 of adjuState. If bit 5 of adjuState is not set this value is old / irrelevant.

3.26.2.5 uint16_t msDif

Actual ms deviation.

This is the current fraction deviation converted to milliseconds (0..999). The sign is Bit 7 of adjuState.

3.26.2.6 uint16_t msDifPrev

Previous ms deviation.

This is the previous fraction deviation converted to milliseconds (0..999). The sign is Bit 3 of adjuState.

The documentation for this struct was generated from the following file:

- include/we-aut_sys/ntp.h

3.27 ntpTimestamp_t Struct Reference

Data Fields

- ucnt32_t fraction

The fractions of the second.

- ucnt32_t seconds

  The seconds.

3.27.1 Detailed Description

NTP time stamp type.

The NTP time stamp type as whole is a 64 bit unsigned number the upper 32 bits being the seconds since era start and the lower 32 bits being their fraction in 2∗∗-32 = ∼0.23ns resolution.

Even being unsigned by declaration differences may be negative appearing as very large numbers. Modulo arithmetic will tread them right in addition and subtraction — only careless comparisons might go wrong.

/note Both parts (and the whole thing) is in wrong (big endian / network) byte order.

3.27.2 Field Documentation

3.27.2.1 ucnt32_t seconds

The seconds.

The zero date for NTP (era 0) is January 1st 1900 (1900-1-1; 1.1.1900) UTC. The wrap around at February 7th 2036 (next "era") is happily defined using 2∗∗32 modulo arithmetic in a way one practically hasn't to worry about.

/note This field is in wrong (big endian / network) byte order.

3.27.2.2 ucnt32_t fraction

The fractions of the second.

The Bits of fraction.v8[3] are to be interpreted as [7|...|0]:

1/2 s | 1/4 s | 1/8 s | 1/16 s | ...

500 ms | 250 ms | 125 ms | ∼62 ms ... The fraction digits 1/16 s and below cannot be expressed in ms. This is a complication for (most) systems using ms (or µs) clocks respectively counters.

The documentation for this struct was generated from the following file:

- include/we-aut_sys/ntp.h

3.28 outFlashTextThr_data_t Struct Reference

Data Fields

- uint16_t indToChar

  Index to first character in first (current) string to be output.
- int8_t noOfFlashStrings2out

  The number of (flash) strings to be output.
- pt_t pt

  The (raw) protothread data structure.
- char const ∗const ∗ theFlashStrings2out

  Pointer to (flash) array of (flash) strings.

3.28.1 Detailed Description

The organisational data for a flash strings array output thread.

See also

outFlashTextThreadF
initOutFlashTextThread

3.28.2 Field Documentation

3.28.2.1 int8_t noOfFlashStrings2out

The number of (flash) strings to be output.

If the original (initialised) value is 1..127 that is the number of 0-terminated flash strings to be output as is (i.e. no return appended) the value will be decremented to 0 in the process if no NULL-pointer in the flash array of flash strings terminates it beforehand.

If the value was initialised to -1 this will act as 1 outputting the 0-terminated flash string pointed to. But additionally, if the bytes after the terminating \0 are \x03 \x01 \x02 (etx, soh, stx) the next byte will be considered as a follow up 0-terminated flash string. Hence by

```
"line1 \n\0\x03\x01\x02line2 \n\0\x03\x01\x02line3 \n\0 no line4"
```

a "pseudo flash string array for \ref outFlashTextThreadF() may be constructed that all others would see as just "line1 \n".

Hint 1: Any other sequence after a terminating 0 will end this continuation.

Hint 2: If the stream used does not provide a putS_P (.putS_P_func) function correctly returning the bytes output, this feature will not work (that is stop with the first "real" string). All weAutSys' streams would work.

3.28.2.2 char const∗ const∗ theFlashStrings2out

Pointer to (flash) array of (flash) strings.

Should be initialised to the string respectively array element to be output first. In the case of multiple string output this pointer will advanced.

3.28.2.3 uint16_t indToChar

Index to first character in first (current) string to be output.

This should (and will normaly) be initialised to 0.

The documentation for this struct was generated from the following file:

- include/we-aut_sys/common.h

3.29 pt Struct Reference

Data Fields

- lc_t lc

  The local continuation.

3.29.1 Detailed Description

A protothread's (raw) data structure.

Examples:

main.c.

The documentation for this struct was generated from the following file:

- pt/pt.h

3.30 smcThr_data_t Struct Reference

Data Fields

- uint8_t bufferState

  The buffer state.
- uint8_t cardType

  The recognised small memory card (SMC)
- uint8_t csd [16]

  The cards CSD.
- uint8_t flag

  The action flag.
- uint8_t lastCmdResp [5]

  Last command response.
- uint8_t lastDatResp

  Last data response.
- uint8_t ocr [4]

  The cards OCR.
- pt_t pt

  The (raw) protothread data structure for the SMC handling thread.
- uint8_t retC

  The return code of the last schedule.
- uint32_t rwSector

  Buffer address.
- uint8_t sectorBuff [SMC_SEC_BUF_SZ]

  The sector buffer.
- p2ptFunV theSMCfun

  The SMC thread function (pointer to)

3.30.1 Detailed Description

The organisational data for a small memory card (SMC) handling thread.

A structure of this type hold the state of one small memory card (SMC) as well as of a handling thread.

See also

appCliThreadF
appCliThread

3.30.2 Field Documentation

3.30.2.1 uint8_t flag

The action flag.

0 means nothing to do.

Bit 4 : power up (80 dummy clocks), CT_POWERD_UP

Bit 5 : set idle CT_INSERTED

Bit 6 : determine card type FL_CAT

Bit 7 : if no card inserted and powered up, check for it.

Bit 1 : read sector FL_RDS

Bit 2 : write sector FL_WRS

Bits 0..6 will be cleared by the respective action performed.

Bit 7 will not be cleared by the "check for card" action. If a card is (newly) detected bits 4 to 6 will be set

3.30.2.2 uint8_t retC

The return code of the last schedule.

This is intended to hold the return code, i.e. the SMC thread function's last return value.

3.30.2.3 p2ptFunV theSMCfun

The SMC thread function (pointer to)

The function pointer will be initialised pointing to smcThreadF, the system provided SMC handling thread. It is provided for future use to put an application specific handler in.

See also

smcThreadF
initSMCthread

3.30.2.4 uint8_t bufferState

The buffer state.

Bit 1 : sector in sync with SMC by just so read or written FL_RDS

Bit 2 : sector to be written after being modified FL_WRS

If neither bit 1 nor bit 2 is set the buffer .sectorBuff has no relation to the SMC's sector `rwSector`. If either is set .sectorBuff is the actual or intended content implying a read operation on `rwSector` redundant.

3.30.2.5 uint8_t lastDatResp

Last data response.

This byte is set by some (block) write / read functions as received from the device. The bit pattern is xxx0rrr1.

rrr = 010 : data accepted

rrr = 101 : data rejected due to a CRC error

rrr = 110 : data rejected due to a write error

3.30.2.6 uint8_t lastCmdResp[5]

Last command response.

In this field the functions sendCmd(), sendCmd0arg() and alike store their last return value for later reference. A R1 response would be in the first byte (SMC_LAST_R1 or lastCmdResp[0]).

3.30.2.7 uint8_t ocr[4]

The cards OCR.

This is the informational part of the R3 response. It is valid if (one bit of) CT_TYPE_MASK bit is set in .cardType

3.30.2.8 uint8_t csd[16]

The cards CSD.

This is the informational part of the R3 response. It is valid if (one bit of) CT_TYPE_MASK bit is set in .cardType

3.30.2.9 uint8_t cardType

The recognised small memory card (SMC)

The value 0 means no usable card recognised. The lower 4 bits contain the card type information; the upper 4 bits show recognised operational conditions

Bit 0 (CT_V_3) : MMC version 3

Bit 1 (CT_V_1) : SD version 1

Bit 2 (CT_V_2) : SD version 2

Bit 3 (CT_HC) : high capacity (V.2) respectively block addressing mode

Bit 4 (CT_POWERD_UP) : card has been successfully powered up (to idle state)

Bit 5 (CT_INSERTED) : a card is (seems to be) inserted

Bit 4 (CT_POWERD_UP) is set by getting a card successfully to idle state. Bit 5 (CT_INSERTED) is set by any valid command response (or card switch). It is reset by no valid card response (i.e. MoSi always tri-stated).

Hint1: Bits 0..4 are compatible to petit FATFS for the purpose of using its file system implementation.

Hint2: For high capacity a 32-bit argument of memory access commands means the block address and the block length is fixed to 512 bytes. (For standard capacity cards this would be a byte address and the block length is determined by CMD16,

3.30.2.10 uint32_t rwSector

Buffer address.

This is the address in the form of a 512 byte sector number (independent of card type) of the buffer .sectorBuff.

3.30.2.11 uint8_t sectorBuff[SMC_SEC_BUF_SZ]

The sector buffer.

This is a buffer for a 512 byte sector. The buffer's size is larger (SMC_SEC_BUF_SZ bytes) to handle some extra bytes like CRC. If the SMC is not used the space can serve other purposes.

If the bit FL_RDS is set in `bufferState` the content is the (cached) SMC's content at `readSector`. If the bit FL_WRS is set in `bufferState` the content is the (cached) SMC's content at `writeSector`.

Any software that changes `writeSector`, `readSector` or the first 512 bytes of `sectorBuff` must clear the respective flag bits in `bufferState`.

See also: FL_CAT FL_RDS FL_WRS

The documentation for this struct was generated from the following file:

- include/we-aut_sys/smc.h

3.31 streamClassDescript_t Struct Reference

Data Fields

- str2CheckOutSpace checkOutSpaceFunc

  This stream's check enough output buffer available function.
- str2InBufferd inBufferdFunc

  This stream's input buffer available function.
- str2OutSpace outSpaceFunc

  This stream's output buffer available function.
- str2PutS_P putS_P_func

  This stream's output a string in flash / program memory function.
- str2PutS putSfunc

  This stream's output a RAM string function.

3.31.1 Detailed Description

Extra stream (type) specific data.

By extensions to the stdio type `FILE` weAutSys will handle textual output and input in a common and extensible way as well as late binding of stream type specific functions.

For additional user / application streams do something like

```
  // have seven specialised functions defined for my extra stream  myStreams
FILE myStreams = FDEV_SETUP_STREAM(myPutChar, myGetChar, _FDEV_SETUP_RW );

struct streamClassDescript_t myStreamsData = {
      .inBufferdFunc = (str2InBufferd)myInBufferd,
      .outSpaceFunc =  (str2OutSpace)myStreamsOutSpace,
      .checkOutSpaceFunc = (str2CheckOutSpace)myCheckOutSpace,
      .putSfunc    = (str2PutS)myPutSt,
      .putS_P_func = (str2PutS_P)myPutSt_P()
};

   // in an initialisation block
   fdev_set_udata(&myStreams, &myStreamsData);
```

That's it. The basic I/O functions `myPutChar` and/or `myGetChar` are, of course, mandatory. All others could be omitted if the respective common / default behaviour is sufficient. For performance reasons those omissions are seldom recommended.

The common behaviour is defined by inBufferd, outSpace, checkOutSpace, putSt and putSt_P. They can be called for the exemplary `myStream` after switching to it.

3.31.2 Field Documentation

3.31.2.1 str2InBufferd inBufferdFunc

This stream's input buffer available function.

Let or set to null if no number in a range 0..buffer size can be determined.

The common function inBufferd() will return AVAILABLE_UNKNOWN then.

3.31.2.2 str2OutSpace outSpaceFunc

This stream's output buffer available function.

Let or set to null if no number in a range 0..buffer size can be determined.

The common function outSpace() will return AVAILABLE_UNKNOWN then.

3.31.2.3 str2CheckOutSpace checkOutSpaceFunc

This stream's check enough output buffer available function.

Let or set to null if this can't never be determined (i.e. the return value would always be 1).

The common function outSpace() will return 1 then.

3.31.2.4 str2PutS putSfunc

This stream's output a RAM string function.

If let or set `NULL` the common function putSt() will use `fputs()`.

3.31.2.5 str2PutS_P putS_P_func

This stream's output a string in flash / program memory function.

If let or set `NULL` the common function putSt_P() will use `fputs_P()`.

The documentation for this struct was generated from the following file:

- include/we-aut_sys/streams.h

3.32 thr_data_t Struct Reference

Data Fields

- union {
  struct cliThr_data_t cliThrData
  Commandline interpreter structure variant.
  struct modThr_data_t modThrData
  Modbus handler variant.
  char stateA [EXTRA_THR_ST_SZ]
  Byte / char array (castable)
  };

  The thread's respectively thread functions state data.
- uint8_t flag

  A flag byte for the threads specific usage.
- p2ptFun fun

  The thread's function (pointer to)
- pt_t pt

  The (raw) protothread data structure.

- uint8_t retC

  The return code of the last schedule.

3.32.1 Detailed Description

State data for a thread (universal variable type)

It is strongly recommended that threads with states larger than fitting in mThr_data_t should use this structure (extending mThr_data_t) for their state. The thread function's (the behaviour) recommended signature takes (only) a pointer to such structure as parameter.

Software using this schema is supported by some handling functions and macros provided.

Note

The first four elements (pt, flag, retC, fun) are exactly the structure mThr_data_t.

Examples:

main.c.

3.32.2 Field Documentation

3.32.2.1 pt_t pt

The (raw) protothread data structure.

This is the same as .pt in mThr_data_t.

Examples:

main.c.

3.32.2.2 uint8_t flag

A flag byte for the threads specific usage.

This is the same as .flag in mThr_data_t.

Examples:

main.c.

3.32.2.3 uint8_t retC

The return code of the last schedule.

This is the same as `retC` in mThr_data_t.

3.32.2.4 p2ptFun fun

The thread's function (pointer to)

This is the same as .fun in mThr_data_t. For compatibility with the bequeather even the type is the same, even though in this larger structure the function pointer put here must be of type p2ptFunC.

3.32.2.5 union { ... }

The thread's respectively thread functions state data.

It is strongly recommended to use no other external or static data for whatever else thread state data but to keep it here in this union. The char [] variant may be cast to any application defined structure of fitting length.

The documentation for this struct was generated from the following file:

- include/we-aut_sys/common.h

3.33 timer_t Struct Reference

Data Fields

- ucnt32_t end

  The timer's end (lapse) time.
- struct timer_t ∗ next

  The next timer.
- ucnt32_t period

  The timer's period.
- uint8_t state

  The timer's state and type.

3.33.1 Detailed Description

The timer data type resp.

structure

This type defines weAutSys' timers of 32 bit resolution in ms or s.

Note

All fields of a timer may be read by application software even if that makes only partial sense. But !
No field of a timer must directly be set by application software.
It's best to use only the timer functions for manipulating timers.

See also

msClock(void)
s_time(void)
TIMER_LAPSED

Examples:

main.c.

3.33.2 Field Documentation

3.33.2.1 ucnt32_t end

The timer's end (lapse) time.

For a ms resolution timer:

This is the end time of a running timer in the (absolute) scale of ref msClock "ms since" system reset / restart (msClock).

End time may be 28,6 days in the future for this timer type.

For a seconds resolution timer:

This is the end time of a running timer in the (absolute) scale of ref secClock "seconds since" system reset / restart (secClock) respectively seconds since March 2008 local time (secTime308Loc()).

End time may be 68 years in the future for this timer type.

See also

TIMER_RUNNING

3.33.2.2 ucnt32_t period

The timer's period.

This is the actual period for the timer's current run or next re-start.

See also

msClock(void)
secClock
TIMER_RUNNING

3.33.2.3 uint8_t state

The timer's state and type.

See also

TIMER_LAPSED
TIMER_RUNNING
TIMER_STOPPED
TIMER_TYPE_FREE
TIMER_UNUSED
ms_TIMER_TYPE
sec_dur_TIMER_TYPE
sec_dat_TIMER_TYPE

3.33.2.4 struct timer_t∗ next

The next timer.

This points to the next timer if this timer is in any system maintained list.

- This is for system software only!

The value must not be modified by user software; see also warnings above.

The only exception is for user / application software that chooses to handle a pool of timers and if the state of a pool timer is TIMER_TYPE_FREE.

The documentation for this struct was generated from the following file:

- include/we-aut_sys/timing.h

3.34 u16div_t Struct Reference

3.34.1 Detailed Description

Two unsigned words intended for quotient and remainder.

This structure is used as return type of divide functions, like divByVal10(uint8_t), divWByVal10toByte(uint16_t) etc. But it can be used as a common pair of unsigned 16 bit values, as well.

See also

u8div_t

The documentation for this struct was generated from the following file:

- D:/eclipCeWS/atMegaBootloader/include/we-aut_sys/ll_common.h

3.35 u8div_t Struct Reference

3.35.1 Detailed Description

Two unsigned bytes intended for quotient and remainder.

This structure is used as return type of divide functions, like divByVal10(uint8_t), divWByVal10toByte(uint16_t) etc. But it can be used as a common pair of unsigned 8 bit values, as well.

See also

u16div_t

The documentation for this struct was generated from the following file:

- D:/eclipCeWS/atMegaBootloader/include/we-aut_sys/ll_common.h

3.36 ucnt16_t Union Reference

Data Fields

- uint16_t v16

  The value as 16 bit unsigned.
- uint8_t v8 [2]

  The values as 8 bit unsigned.
- uint16_t val

  The (full) value (16 bit unsigned)

3.36.1 Detailed Description

A medium (16 bit) value in different resolutions.

This type is for any (non negative) counting or other 16 bit value.

It features 8 and 16 bit resolution as well as direct access to the high and low byte.

See also

ucnt32_t

Examples:

main.c.

3.36.2 Field Documentation

3.36.2.1 uint8_t v8[2]

The values as 8 bit unsigned.

`v8`[0] would be the 8 bit counter value — in architecture (Atmel little) endianess. And `v8`[1] would be the high byte.

If `v16` is used to hold a network (wrong, big) endian value that would be the other way round, of course.

Examples:

main.c.

The documentation for this union was generated from the following file:

- D:/eclipCeWS/atMegaBootloader/include/we-aut_sys/ll_common.h

3.37 ucnt32_t Union Reference

Data Fields

- uint16_t v16 [2]

  The value as 16 bit unsigned.
- uint32_t v32

  The value as 32 bit unsigned.
- uint8_t v8 [4]

  The values as 8 bit unsigned.
- uint32_t val

  The full value (32 bit unsigned)

3.37.1 Detailed Description

A big (32 bit) value in different resolutions.

This type is for any (non negative) counting or other 32 bit value or for use as a result "accumulator", even in "wrong" big endianess.

It features 8, 16 and 32 bit resolution as well as direct access to all four bytes.

See also

msClock(void)
ucnt16_t

3.37.2 Field Documentation

3.37.2.1 uint16_t v16[2]

The value as 16 bit unsigned.

Due to unsigned arithmetic properties it is quite simple to base all millisecond resolution timers up to ∼32s on this resolution.

3.37.2.2 uint8_t v8[4]

The values as 8 bit unsigned.

v8[0] would be the 8 bit counter value — in architecture (atmel little) endianess.

The documentation for this union was generated from the following file:

- D:/eclipCeWS/atMegaBootloader/include/we-aut_sys/ll_common.h

3.38 uip_eth_addr Struct Reference

3.38.1 Detailed Description

Representation of a 48-bit Ethernet address / MAC address.

The documentation for this struct was generated from the following file:

- include/uip/uip.h

3.39 uip_eth_hdr Struct Reference

Data Fields

- struct uip_eth_addr dest

  source 48 bit MAC address
- struct uip_eth_addr src

  destination 48 bit MAC address
- uint16_t type

  The (big endian) type if $>$ 1500.

3.39.1 Detailed Description

The Ethernet header.

The documentation for this struct was generated from the following file:

- include/uip/uip_arp.h

3.40 uip_icmpip_hdr Struct Reference

3.40.1 Detailed Description

The ICMP and IP headers.

The documentation for this struct was generated from the following file:

- include/uip/uip.h

3.41 uip_tcpip_hdr Struct Reference

3.41.1 Detailed Description

The TCP and IP headers.

The documentation for this struct was generated from the following file:

- include/uip/uip.h

3.42 uip_udpip_hdr Struct Reference

3.42.1 Detailed Description

The UDP and IP headers.

The documentation for this struct was generated from the following file:

- include/uip/uip.h

3.43 uipConn_t Struct Reference

Data Fields

- uip_tcp_appstate_t appstate

  The application state.
- uint16_t initialmss

  Initial maximum segment size for the connection.
- uint16_t len

  Length of the data that was previously sent.
- uint16_t lport

  The local TCP port, in network byte order.
- uint16_t mss

  Current maximum segment size for the connection.
- uint8_t nrtx

  The number of retransmissions for the last segment sent.
- uint8_t rcv_nxt [4]

  The sequence number that we expect to receive next.
- uint16_t readIndex

  for the LAN (stream) input functions
- uip_ipaddr_t ripaddr

  The IP address of the remote host.
- uint16_t rport

  The remote TCP port, in network byte order.
- uint8_t rto

  Retransmission time-out.

- uint8_t sa

  Retransmission time-out calculation state variable.
- uint8_t snd_nxt [4]

  The sequence number that was last sent by us.
- uint8_t sv

  Retransmission time-out calculation state variable.
- uint8_t tcpstateflags

  TCP state and flags.
- uint8_t timer

  The retransmission timer.
- uint16_t writeIndex

  for the LAN (stream) output functions

3.43.1 Detailed Description

Representation of a uIP TCP connection.

The uip_conn structure is used for identifying a connection. All but one field in the structure are to be considered read-only by application software. The only exception is the field `appstate` whose purpose is to let the application store application-specific state for the connection in a structure of appropriate type (e.g. cliThr_data_t or ntpState_t) not longer than the specified specified maximum size.

The documentation for this struct was generated from the following file:

- include/uip/uip.h

Chapter 4

File Documentation

4.1 D:/eclipCeWS/atMegaBootloader/bootloader/boot109.c File Reference

4.1.1 Overview

Implementation of weAutSys' serial bootloader program. weAutSys comes with a serial bootloader according to Atmel Corporation application note AVR109 on "Self-programming".

This bootloader is (as of end 2014) implemented for and tested on four different target platforms respectively µ-Controllers:

- ATmega1284P as used on the weAut_01 automation module with runtime weAutSys'
- ATmega2560 on ArduinoMega2560 or on ArduinoMegaADK etc.
- ATmega328p as used on ArduinoUno
- ATmega32 as shipped e.g. with easyAVR V.7 and BoAVR2013 evaluation boards

By C macro control and by Make support this implementation covers said platforms respectively µControllers.

Besides to those targets this serial bootloader can easily be adapted to all similar platforms respectively to all AVR 8 bit ATmega µCs featuring self-programming capability.

In this implementation the serial bootloader's baudrate is defaults to

1. This setting yields quite a good performance as well as good transmission quality. In a way 38400 is the best value for serial boot-loading. Using lower rates would throw performance away, while higher baud-rates might require more buffering as flashing would get much slower than serial transmission in some cases.

   Regarding the transmission quality 115200 was tested successfully on (real) RS232 links and with excellent cabling conditions and very accurate clock frequency generators.

An exception to the 38400 baud rule is the easyAVR evaluation board. It is shipped with a 8 MHz quartz for the CPU clock generator, even if the ATmega32 could use 16 MHz. 8 MHz gives poor accuracy for 38400 baud; hence we use 19200 as standard (set in the respective includes) for this platform.

All target platform's specific settings are organised in include files for both the C-Compiler and the make-tool. Please read those files for the respective platform and on how to extend this project to other hardware targets.

For technical background information on serial bootloading for ATmegas and this bootloader please read `a-weinert.de/pub/AVRserBootl.pdf`.

Revision:

1

Date:

2017-01-25 12:03:06 +0100 (Mi, 25 Jan 2017)

Functions

- void basicSystemInit ()

  Initialise system resources.
- void bootLoaderGreet (void)

  Send the greeting lines for bootloader's start.
- int main (void)

  The bootloader's start.

Variables

- char const bootRevDat []

  The bootloader's revision and date.
- char const greetByCLI []

  Greeting text for the case of being entered by CLI.
- char const programmerHWver []

  The programmers hardware version.
- char const programmerSign []

  Software identifier/programmer signature.
- char const programmerSWver []

  The programmers software version.

4.1.2 Variable Documentation

4.1.2.1 char const **greetByCLI[ ]**

Greeting text for the case of being entered by CLI.

If the bootloader was entered by a command line interpreter (CLI) due to human command via UART, a greeting response may be sent via that serial line extended by below hints on (AVR109) bootloader commands.

This response will only be sent if the macro ALLOW_UART_GREET_BOOTLOADER_BY_CLI is defined with a value !=0. This setting if usually done in a platform specific include file.

4.2 D:/eclipCeWS/atMegaBootloader/bootloader/bootLib.c File Reference

4.2.1 Overview

Implementation ofweAutSys' bootloader helper functions. weAutSys comes with a serial bootloader according to Atmel Corporation application note AVR109 on "Self-programming".

This bootloader was implemented for and tested on some quite different target platforms, ref intro_secH "weAut_01" being on of them. It can easily be adapted to all similar platforms respectively to all AVR µCs with self-programming support.

One development goal was the tight integration with the system / application software. That means

a) not re-doing all or part of the platform initialisation and

b) re-using bootloader functions in application software,

c) which can also be done for final (flash) bootloader variables.

See file boot109.c and

Albrecht Weinert

```
A serial bootloader for weAut_01, ArduinoMega and akin
```

Revision:

1

Date:

2017-01-25 12:03:06 +0100 (Mi, 25 Jan 2017)

Functions

- void appMain (uint8_t init)

  Jump to application program.
- ADDR_T blockLoadEE (uint16_t size, ADDR_T address)

  Write a sequence of bytes read from UART(0) to EEPROM.
- ADDR_T blockLoadFl (uint16_t size, ADDR_T address)

  Write a sequence of words read from UART(0) to flash memory.
- ADDR_T blockReadEE (uint16_t size, ADDR_T address)

  Read a sequence of bytes from EEPROM and send them via UART(0)
- ADDR_T blockReadFl (uint16_t size, ADDR_T address)

  Read a sequence of bytes from flash memory and send them via UART(0)
- void bootMain (void)

  Jump to the bootloader.
- char ∗ copyChars_P (char ∗dst, ADDR_T src, uint8_t mxLen)

  Copy a string from flash memory to RAM.
- uint16_t div16 (uint16_t ∗rem, uint16_t dividend, uint16_t divisor)

  Unsigned 16 bit divide.
- uint32_t div24 (uint32_t ∗rem, uint32_t dividend, uint32_t divisor)

  Unsigned 24 bit divide.
- uint32_t div32 (uint32_t ∗rem, uint32_t dividend, uint32_t divisor)

  Unsigned 32 bit divide.
- uint32_t div32by16 (uint16_t ∗rem, uint32_t dividend, uint16_t divisor)

  Unsigned 32 bit divide by 16 bit.
- uint32_t div32by24 (uint32_t ∗rem, uint32_t dividend, uint32_t divisor)

  Unsigned 32 bit divide by 24 bit.
- void initUART0 (uint32_t baudRate, uint8_t x2, uint8_t parity, uint8_t stopBits, uint8_t useInt)

  Initialise the serial input (UART0)
- uint8_t isFlashCleared (void)

  Flash memory has no program.
- uint8_t recvErrorState (void)

  Get UART receive error status and flush receiver on error.
- uint8_t recvSerByte (void)

  Basic UART receive one byte.
- ADDR_T resetCauseText_P (uint8_t resetCauses)

  The reset cause text.
- void sendSerByte (uint8_t c)

  Basic UART send one byte (guarded)
- void sendSerBytes (char ∗src)

  Basic UART0 send multiple bytes from RAM.

- void sendSerBytes_P (ADDR_T src)

  Basic UART send multiple bytes from flash.
- void setTheLed (uint8_t state)

  Set the LED.
- void toHMI8LEDchain (uint8_t val)

  Output to a chain of eight HMI/visible LEDs.
- uint32_t uartBaud (uint16_t uartPresc, uint8_t x2)

  Calculate the true baudrate for a prescaler setting.
- uint16_t uartPrescaler (uint32_t baudRate, uint8_t x2)

  Calculate the prescaler setting for a desired baudrate.
- void wait25 (void)

  A very basic delay function keeping the CPU busy for about 25µs.
- uint8_t waitSerByte (uint8_t tOut)

  Basic UART wait for a byte received.

Variables

- char const bootAut []

  The author of weAutSys and its bootloader.
- char const bootCop []

  The copyright notice for weAutSys and its bootloader.
- char const bootloaderPlatf []

  Bootloader's platform name and CPU frequency.
- char const bootloaderWlc []

  Bootloader's welcome greeting and copyright notice.
- char const bootResetCause0 []

  no reset cause: exit from from active bootloader or by command
- char const bootResetCause1 []

  reset cause: power on
- char const bootResetCause2 []

  reset cause: external
- char const bootResetCause4 []

  reset cause: brown out
- char const bootResetCause8 []

  reset cause: watchdog
- char const bootResetCauseG []

  reset cause: JTAG
- char const bootResetCauseN []

  reset cause: not known
- char const bootTextReset []

  The text "Reset by:".
- char const bootTextRevis []

  The text "Revision:".
- char const greetEmptFlash []

  Greeting for empty application flash.
- char const systBld []

  The build date and time.
- uint8_t const timOutCountPatr []

  An eight bit count down display pattern.

4.3 D:/eclipCeWS/atMegaBootloader/include/arch/config.h File Reference

4.3.1 Overview

weAutSys' configuration settings This file contains some definitions concerning software and hardware configuration. These settings influence the compilation and building process and can't be changed later at runtime.

This file (in the serial bootloader project) is part of weAutSys <weinert-automation.de>

It is also used for the serial bootloader.

Mainly by an include file mechanism (config_<platformName>.h) a number of different (AVR-) platforms are supported.

Copyright © 2014 Albrecht Weinert, Bochum

Author

Albrecht Weinert <a-weinert.de>

Revision:

1

Date:

2017-01-25 12:03:06 +0100 (Mi, 25 Jan 2017)

Defines

- #define GN_LED_INV

  invert gn LED: ! ; no invert: empty
- #define RD_LED_INV

  invert rd LED: ! ; no invert: empty
- #define STD_BAUD 38400

  Standard resp. used baudrate.
- #define TRAMPOLIN_USE 1

  The trampolin hack is used.
- #define USART_IN_APP_WITH_INT

  UART uses no interrupt in application.
- #define USE_BOOTLOADER USE_BOOTLOADER_PRESET

  Have and use a bootloader.

4.4 D:/eclipCeWS/atMegaBootloader/include/arch/config_weAut_01.h File Reference

4.4.1 Overview

Configuration settings for the weAut_01 automation module. This file contains some platform specific definitions concerning software and hardware configuration. These settings influence the compilation and build process. Most of those settings can't be changed later at runtime.

Platform: weAut_01 (weAut_00); CPU: ATmega1284p;

Hardware specifica and settings:

Port A: DI or AI channels

Port B B0 in enter key in pullup 1 B1 in RTS in / counter in / free port JP9 (default in) 1 B2 out LED gn (0 = on) 0 B3 in comparator LV (Lastspannung) 0 B4 out LED rd (1 = on) 1 B7,6,5 SPI1-Master: B7clk, B5mosi must be set as output 101 Port B out 2 4 (LEDs) 5 7 SPI init SPI 1 : used for DI_LEDS and DO Enable SPI, Master, Int (SPR1,0=00 : clock /4) SPI 1 runs with 1/2 CPU clock, ie. 10MHz, needing 16 clocks for one byte

Port C: C2, C3, C6: /CS=Clk : out 1; C5 : DO enab : out 0; 0: DO disable (LED red) C7 : ether /CS : out 1; ether /reset (out 0) on weAut_00 C4 : DO OK : in 1 pullup C01 : Testpin 12 : 1 pullup

Port D: D7 NIC interrupt : in pullup D6 CTS generator : out (default) D5 free port on weAut_01 respectively SMC insertion switch D4 3 2 SPI 2 MOSI MISO Clk : 4&3 out D1 0 UART 0 TX RX : 1 out

This file is part of weAutSys <weinert-automation.de>

It is also used for the serial bootloader.

Copyright © 2014 Albrecht Weinert, Bochum

Author

Albrecht Weinert <a-weinert.de>

Revision:

1

Date:

2017-01-25 12:03:06 +0100 (Mi, 25 Jan 2017)

Defines

- #define ALLOW_UART_GREET_BOOTLOADER_BY_CLI 1

  Report bootloader entrance by CLI on UART.
- #define BASIC_INIT()

  Port initialisation.
- #define BUZZER_PIN 1

  Pin for buzzer (if any)
- #define BUZZER_PORT C

  Port for buzzer (if any)
- #define GN_LED_PIN 2

  Pin for green Status LED.
- #define GN_LED_PORT B

  Port for green Status LED.
- #define HAVE_ETHERNET 1

  Ethernet available.
- #define HAVE_MODBUS 1

  Modbus on Ethernet available.
- #define HAVE_NIC 2

  Network interface chip.
- #define HAVE_SMC 2

  Small memory card via SPI.
- #define PROG_HW_VER

  The programmerHWver.
- #define RD_LED_PIN 4

  Pin for red Status LED.
- #define RD_LED_PORT B

  Port for red Status LED.
- #define SMC_INS_POLL

  SMC insert check by software thread.

- #define SMC_INSERT 0xD5

  SMC insert switch; 0: no switch.
- #define TIMEOUT_LIGHTSHOW

  Time out HMI display.
- #define UART0MAY_CTS 1

  UART0 may signal ability to receive by CTS.
- #define UART0MAY_RTS 1

  UART0 may use RTS to hold back sending.
- #define UART0SPIYKY 0

  UART0 has spiky USB2serial: filter FE,FF.
- #define USART_IN_APP_WITH_INT

  UART uses interrupt in application.
- #define USE_BOOTLOADER_PRESET

  Have and use a bootloader.
- #define USE_SPI2 1

  Use SPI2 for NIC and SMC.

4.4.2 Define Documentation

4.4.2.1 #define ALLOW_UART_GREET_BOOTLOADER_BY_CLI 1

Report bootloader entrance by CLI on UART.

Setting this to 1 we assume the UART on this platform being used for textual human interaction including command line interpreter (CLI).

4.4.2.2 #define SMC_INS_POLL

SMC insert check by software thread.

If evaluated to non zero the SMC thread checks the card insertion by software at otherwise idle times about every 100 ms. This is not necessary if a hardware insert switch is available and properly configured.

The insert test by software is not recommended if every SMC command is logged by DEBUG_SMC $>$ 1.

4.4.2.3 #define BASIC_INIT( )

Port initialisation.

This macro does the basic port initialisation according to the platform's standard port usage. Anyway it has to cover the serial bootloader.

4.4.2.4 #define USE_BOOTLOADER_PRESET

Have and use a bootloader.

This value determines the existence and usage of a bootloader:

0: no bootloader or at least no bootloader integration 1: bootloader area used / bootloader existing 2: bootloader fully integrated: application programme may use all No Read-While-Write (NRWW) flash section functions and constants.

Fully integrated (2) means that all µController resets go through the bootloader. And the bootloader will not enter the program (=x00000) without having done all basic initialisations, that hence can be omitted from system/application software. Full integration also means that the bootloader may be entered by (CLI) command.

4.5 D:/eclipCeWS/atMegaBootloader/include/bootloader/boot109.h File Reference

4.5.1 Overview

Definitions of weAutSys' bootloader and helper functions. These functions are to be used in situations before or after a reset or a system re-initialisation, that is without any threading support (or interrupts). This condition is met

- in early initialisation phases of weAutSys' system or application software and
- at all the time in the bootloader.

Everything defined here is to be linked and put to the NMWR (no modify while read) respectively bootloader section of the flash/program memory. It is usable by normal system and application software through certain linkage mechanisms described in in Albrecht Weinert

`A serial bootloader for weAut_01, ArduinoMega and akin`

Copyright © 2015 Albrecht Weinert, Bochum

Author

Albrecht Weinert <`a-weinert.de`>

Revision:

1

Date:

2017-01-25 12:03:06 +0100 (Mi, 25 Jan 2017)

History Rev. 8..13 2014-10-16 : stable _ext.tab; check this out for clients linked threreto Rev. 14 2014-10-17 .. : library cleanup; separate "light show" from serial I/O functions

Defines

- #define FAR_ADD(varHF)

  Generate a far address for high flash memory items.
- #define getFlashByte(addr)

  Read a flash byte also from high addresses.
- #define getSerByte()

  Basic function: UART(0) get one byte already received.
- #define hwCritEntBoot()

  Hardware criterion for entering bootloader after reset.
- #define isSerByteRec()

  Basic function: UART(0) has one byte received.
- #define LOW_ADD(varLF)

  Generate a far address for low flash memory items.
- #define PARTCODE 0x44

  The (outdated) single byte part code.
- #define PROG_HW_VER

  The programmerHWver.
- #define PROG_SIGN "AVRBOOT"

  The programmerSign.
- #define PROG_SW_VER "10"

  The programmerSWver.

- #define SEC_with_F0 (72-61)

  timeout value for timOutCountPatr
- #define SEC_with_F8 (75-56)

  timeout value for timOutCountPatr
- #define SEC_with_FF (72-35)

  timeout value for timOutCountPatr
- #define TIMEOUT_LIGHTSHOW

  Time out HMI display.

Functions

- void appMain (uint8_t init) __attribute__((noreturn))

  Jump to application program.
- void basicSystemInit (void)

  Initialise system resources.
- ADDR_T blockLoadEE (uint16_t size, ADDR_T address)

  Write a sequence of bytes read from UART(0) to EEPROM.
- ADDR_T blockLoadFl (uint16_t size, ADDR_T address)

  Write a sequence of words read from UART(0) to flash memory.
- ADDR_T blockReadEE (uint16_t size, ADDR_T address)

  Read a sequence of bytes from EEPROM and send them via UART(0)
- ADDR_T blockReadFl (uint16_t size, ADDR_T address)

  Read a sequence of bytes from flash memory and send them via UART(0)
- void bootLoaderGreet (void)

  Send the greeting lines for bootloader's start.
- void bootMain (void) __attribute__((noreturn))

  Jump to the bootloader.
- char ∗ copyChars_P (char ∗dst, ADDR_T src, uint8_t mxLen)

  Copy a string from flash memory to RAM.
- void initUART0 (uint32_t baudRate, uint8_t x2, uint8_t parity, uint8_t stopBits, uint8_t useInt)

  Initialise the serial input (UART0)
- uint8_t isFlashCleared (void)

  Flash memory has no program.
- uint8_t recvErrorState (void)

  Get UART receive error status and flush receiver on error.
- uint8_t recvSerByte (void)

  Basic UART receive one byte.
- ADDR_T resetCauseText_P (uint8_t resetCauses)

  The reset cause text.
- void sendSerByte (uint8_t c)

  Basic UART send one byte (guarded)
- void sendSerBytes (char ∗src)

  Basic UART0 send multiple bytes from RAM.
- void sendSerBytes_P (ADDR_T src)

  Basic UART send multiple bytes from flash.
- void setTheLed (uint8_t state)

  Set the LED.
- void toHMI8LEDchain (uint8_t val)

  Output to a chain of eight HMI/visible LEDs.
- void wait25 (void)

  A very basic delay function keeping the CPU busy for about 25µs.
- uint8_t waitSerByte (uint8_t tOut)

  Basic UART wait for a byte received.

Variables

- char const bootAut []

  The author of weAutSys and its bootloader.
- char bootBld []

  The build date and time.
- char const bootCop []

  The copyright notice for weAutSys and its bootloader.
- char const bootloaderPlatf []

  Bootloader's platform name and CPU frequency.
- char const bootloaderWlc []

  Bootloader's welcome greeting and copyright notice.
- char const bootResetCause0 []

  no reset cause: exit from from active bootloader or by command
- char const bootResetCause1 []

  reset cause: power on
- char const bootResetCause2 []

  reset cause: external
- char const bootResetCause4 []

  reset cause: brown out
- char const bootResetCause8 []

  reset cause: watchdog
- char const bootResetCauseG []

  reset cause: JTAG
- char const bootResetCauseN []

  reset cause: not known
- char const bootRevDat []

  The bootloader's revision and date.
- char const bootTextReset []

  The text "Reset by:".
- char const bootTextRevis []

  The text "Revision:".
- char const greetEmptFlash []

  Greeting for empty application flash.
- char const programmerHWver []

  The programmers hardware version.
- char const programmerSign []

  Software identifier/programmer signature.
- char const programmerSWver []

  The programmers software version.
- uint8_t const timOutCountPatr []

  An eight bit count down display pattern.

4.6 D:/eclipCeWS/atMegaBootloader/include/we-aut_sys/ll_common.h File Reference

4.6.1 Overview

weAutSys' basic common types and helper functions This file contains the definitions for weAutSys' common types and helper macros or functions. They are utilised in more than one application field respectively application or system module as well as in the serial bootloader.

This file is part of weAutSys <weinert-automation.de>

It is also used for the serial bootloader.

Copyright © 2013 Albrecht Weinert, Bochum

Author

Albrecht Weinert <a-weinert.de>

Revision:

1

Date:

2017-01-25 12:03:06 +0100 (Mi, 25 Jan 2017)

Data Structures

- struct u16div_t

  Two unsigned words intended for quotient and remainder.
- struct u8div_t

  Two unsigned bytes intended for quotient and remainder.
- union ucnt16_t

  A medium (16 bit) value in different resolutions.
- union ucnt32_t

  A big (32 bit) value in different resolutions.

Defines

- #define ADDR_T

  flash address is 16 bit for this µController
- #define ADDR_U

  flash address is 16 bit for this µController
- #define addr_v

  flash address is 16 bit for this µController
- #define andAP(mask)

  Unset (boolean and operation) port bits.
- #define andBP(mask)

  Unset (boolean and operation) port bits.
- #define andCP(mask)

  Unset (boolean and operation) port bits.
- #define andDP(mask)

  Unset (boolean and operation) port bits.
- #define andOrBP(andMask, orMask)

  Unset and set port B bits.
- #define andOrCP(andMask, orMask)

  Unset and set port C bits.
- #define andPort(port, mask)

  Unset (boolean and operation) port-bits.
- #define APP_END

  The end of the application flash area (as byte address + 1)
- #define BEL

  The bell code.

- #define BOOTL_BEG

  The start address of the bootloader (area)
- #define BOOTSIZE 4096

  The (maximum) boot size in bytes.
- #define BS

  The back space code.
- #define CR

  The carriage return code.
- #define dirPort(port)

  Direction port (by letter)
- #define EEP_SIZE (E2END + 1)

  The EEPROM size in byte.
- #define ESC

  The Escape code.
- #define FCPU_S

  The CPU clock frequency (as string)
- #define FF

  The form feed code.
- #define fromPort(port)

  Re-read output from a port (by letter)
- #define HBYTE(x)

  Get the high byte (of a 16 bit value)
- #define HT

  The horizontal tab code.
- #define inPort(port)

  Input port (by letter)
- #define inpPins(port, mask)

  Set port pins as input.
- #define LARGE_MEMORY 0

  The flash memory byte address type.
- #define LBYTE(x)

  Get the low byte (of a 16 bit value)
- #define LF

  The linefeed code.
- #define OFF

  A boolean false respectively off.
- #define ON

  A boolean true respectively on.
- #define orAP(mask)

  Set (boolean or operation) port-bits.
- #define orBP(mask)

  Set (boolean or operation) port-bits.
- #define orCP(mask)

  Set (boolean or operation) port-bits.
- #define orDP(mask)

  Set (boolean or operation) port-bits.
- #define orPort(port, mask)

  Set (boolean or operation) port-bits.
- #define outPins(port, mask)

  Set port pins as output.
- #define PLATFORM_S

The platform name (as string)

- #define setPort(port, value)

  Set all port-bits.
- #define SYST_AUT "Albrecht Weinert <a-weinert.de> "

  The runtime's author.
- #define SYST_BLD

  The runtime's system build and time.
- #define SYST_COP

  The runtime's copyright notice.
- #define TOKENPASTE(x, y)

  Concatenation helper macro x ## y.
- #define toPort(port, mask)

  Output to a port (by letter)
- #define VT

  The vertical tab code.
- #define xorPort(port, mask)

  Toggle (boolean xor operation) port-bits.

Defines due to tools

- #define INFLASH(decl)

  Alternative declaration for flash memory.
- #define INEEPROM(decl)

  Declaration for EEPROM memory.
- #define STR(leMac)

  A macro value as string.

Functions

- uint32_t uartBaud (uint16_t uartPrescal, uint8_t x2)

  Calculate the true baudrate for a prescaler setting.
- uint16_t uartPrescaler (uint32_t baudRate, uint8_t x2)

  Calculate the prescaler setting for a desired baudrate.

Optimised Divide functions (optionally in bootloader)

- uint16_t div16 (uint16_t ∗rem, uint16_t dividend, uint16_t divisor)

  Unsigned 16 bit divide.
- uint32_t div24 (uint32_t ∗rem, uint32_t dividend, uint32_t divisor)

  Unsigned 24 bit divide.
- uint32_t div32 (uint32_t ∗rem, uint32_t dividend, uint32_t divisor)

  Unsigned 32 bit divide.
- uint32_t div32by24 (uint32_t ∗rem, uint32_t dividend, uint32_t divisor)

  Unsigned 32 bit divide by 24 bit.
- uint32_t div32by16 (uint16_t ∗rem, uint32_t dividend, uint16_t divisor)

  Unsigned 32 bit divide by 16 bit.

4.7 include/arch/uip-conf.h File Reference

4.7.1 Overview

IP configuration file. Configuration definitions given in this file are (indirectly) used to define uIP configuration values.

Adam Dunkels' original copyright notice

Copyright (c) 2006, Swedish Institute of Computer Science. All rights reserved.

(contained in this file) still holds. For modifications:

Copyright © 2011 Albrecht Weinert, Bochum

Author

Adam Dunkels adam@dunkels.com
Albrecht Weinert <a-weinert.de>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Project-specific configuration options

uIP has a number of configuration options that can be overridden for each project.

These are kept in a project-specific uip-conf.h file and all configuration names have the prefix UIP_CONF.

- #define UIP_CONF_UDP_CONNS 10

  Maximum number of UDP connections.
- #define UIP_CONF_MAX_CONNECTIONS 10

  Maximum number of TCP connections.
- #define UIP_CONF_MAX_LISTENPORTS

  Maximum number of listening TCP ports.
- #define UIP_CONF_ARPTAB_SIZE 12

  The size of the ARP table.
- #define UIP_CONF_BUFFER_SIZE 620

  uIP buffer size
- #define UIP_CONF_BYTE_ORDER UIP_LITTLE_ENDIAN

  CPU byte order.
- #define UIP_CONF_LOGGING

  Logging on or off.
- #define UIP_CONF_UDP

  UDP support on or off.
- #define UIP_CONF_UDP_CHECKSUMS 1

  UDP checksums on or off.
- #define UIP_CONF_INC_CHECKSUMS 0

  Check incoming checksums on or off.
- #define UIP_CONF_STATISTICS 0

  uIP statistics on or off
- #define **UIP_CONF_BROADCAST**
- #define UIP_APPCALL

Macro to name the uIP event function.

- #define UIP_UDP_APPCALL

  Macro to name the uIP udp event function.
- typedef uint8_t uip_tcp_appstate_t [SIZE_OF_BIGGEST_APPSTATE]

  The type of appstate in uip_conn.
- void uip_appcall (void)

  The uIP event function.
- void udp_appcall (void)

  The uIP udp event function.

4.8 include/arch/uip_mess.h File Reference

4.8.1 Overview

weAutSys definition of system message texts for uIP This is system software and must not be modified for user or application programs.

This file is part of weAutSys <`weinert-automation.de`>

weAutSys puts all fixed uIP message texts in flash memory.

Copyright © 2011 Albrecht Weinert, Bochum

Author

Albrecht Weinert <`a-weinert.de`>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Variables

- char icmpNotIcmpEcho [ ]

  icmp: not icmp echo. (in flash)
- char icmpUnknownMess [ ]

  icmp: unknown ICMP message. (in flash IPV6)
- char ipBadCheckS [ ]

  ip: bad checksum. (in flash)
- char ipFragmDropp [ ]

  ip: fragment dropped. (in flash)
- char ipInvVersHeadL [ ]

  ip: invalid version or header length. (in flash)
- char ipNotTcpNorIcmp [ ]

  ip: neither tcp nor icmp. (in flash)
- char ipNotTcpNorIcmp6 [ ]

  ip: neither tcp nor icmp6. (in flash IPV6)
- char ipPackDroppSiAA [ ]

  ip: packet dropped since no address assigned. (in flash)
- char ipPackShorter [ ]

  ip: packet shorter than reported in IP header. (in flash)

- char ipPossPingConfR []

  ip: possible ping config packet received. (in flash)
- char tcpBadCheckS []

  tcp: bad checksum. (in flash)
- char tcpGotResetAbrt []

  tcp: got reset, aborting connection. (in flash)
- char tcpNoUUsedCon []

  tcp: found no unused connections. (in flash)
- char udpBadCheckS []

  udp: bad checksum. (in flash)
- char udpNoMatchCon []

  udp: no connection matching (in flash)

4.9 include/fatFS/diskio.h File Reference

4.9.1 Overview

Small memory card (SMC / MMC) driver function definitions. This is the basic include file for ChaN's (grande) fatFS. It defines the low level disk / media I/O driver functions.

Copyright (c) 2012, ChaN, all right reserved.

Modification for weAutSys by A. Weinert

Copyright © 2012 Albrecht Weinert

weinert - automation, Bochum

weinert-automation.de weAut.de a-weinert.de

A.W.'s modifications for weAutSys are

- adaptions to ATmel ATmega1284P, i.e. weAut_01 and similar small (petit) systems,
- performance improvements and removal of compiler warnings
- make usable on non pre-emptive (Protothreads based) run-time, i.e weAutSys
- documentation enhancements, mainly by adding (Doxygen) documentation to the code

See the notes on copyright and modifications in file fatFS/ff.h.

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Defines

- #define _USE_IOCTL 1

  enable disk_ioctl function
- #define _USE_WRITE 1

  enable disk_write function
- #define CT_BLOCK 0x08

  Card type flag: block addressing.

- #define CT_MMC 0x01

  Card type flag: MMC version 3.
- #define CT_SD1 0x02

  Card type flag: SD version 1.
- #define CT_SD2 0x04

  Card type flag: SD version 2.
- #define CT_SDC

  Card type mask: SD.
- #define CTRL_ERASE_SECTOR 4

  Generic command: force erased a block of sectors (for only _USE_ERASE)
- #define CTRL_POWER 5

  Generic command: get/set power status.
- #define DRESULT

  Results of disk (media driver) functions.
- #define GET_BLOCK_SIZE 3

  Generic command: get erase block size (for only f_mkfs())
- #define GET_SECTOR_COUNT 1

  Generic command: get media size (for only f_mkfs())
- #define GET_SECTOR_SIZE 2

  Generic command: get sector size.
- #define MMC_GET_CID 12

  Specific ioctl command: get CID.
- #define MMC_GET_CSD 11

  Specific ioctl command: get CSD.
- #define MMC_GET_OCR 13

  Specific ioctl command: get OCR.
- #define MMC_GET_SDSTAT 14

  Specific ioctl command: get SMC status.
- #define MMC_GET_TYPE 10

  Specific ioctl command: get card type.
- #define RES_ERROR 1

  Any (hard) error occurred.
- #define RES_NOTRDY 3

  The drive has not been initialised.
- #define RES_PARERR 3

  Invalid function argument.
- #define STA_NODISK 0x02

  No medium in the drive / slot.
- #define STA_NOINIT 0x01

  Drive not initialized.
- #define STA_PROTECT 0x04

  Medium is write protected by hardware switch.

Functions

- uint8_t disk_initialize (uint8_t drv)

  Initialise the SMC drive.
- DRESULT disk_ioctl (uint8_t drv, uint8_t ctrl, uint8_t ∗buff)

  Special control functions.
- DRESULT disk_read (uint8_t drv, uint8_t ∗buff, uint32_t sector)

Read a sector.

- uint8_t disk_status (uint8_t drv)

  Get drive status.

- DRESULT disk_write (uint8_t drv, const uint8_t ∗buff, uint32_t sector)

  Write a sector.

4.10 include/fatFS/ff.h File Reference

4.10.1 Overview

(Grande) fatFS configurations and declarations This is the main include file for ChaN's (grande) fatFS for small memory cards (SMC, MMC) and FAT file system functions. They are build upon the low level driver functions defined in file fatFS/diskio.h.

Copyright (c) 2012, ChaN, all right reserved.

The original copyright notice is at the begin of the (this) source file fatFS/ff.h. It is still valid.

Modification for weAutSys by A. Weinert

Copyright © 2012 Albrecht Weinert

weinert - automation, Bochum

weinert-automation.de weAut.de a-weinert.de

A.W.'s modifications for weAutSys are

- adaptions to ATmel AATmega1284P, i.e. weAut_01 and similar small (petit) systems,
  - restriction to SPI with small memory cards (SMCs)
  - disallow multi-sector access for SMCs on SPI
  - restriction to small file names at present for assumed license reasons
  - removal of file lock feature
- performance improvements, mending of compiler warnings (bugs) etc.
- make usable on non pre-emptive (Protothreads based) run-time, i.e weAutSys
- documentation enhancements, mainly by adding (Doxygen) documentation to the code

Data Structures

- struct DIR

  Directory object structure (DIR)

- struct FATFS

  File system object structure (FATFS)

- struct FIL

  File object structure (FIL)

- struct FILINFO

  File status structure (FILINFO)

Defines

- #define _FS_TINY

    Tiny FS.
- #define _MAX_SS 512

    Maximum sector size to be handled.
- #define AM_ARC 0x20

    directory entry attribute bit: archive
- #define AM_DIR 0x10

    directory entry attribute bit: directory
- #define AM_HID 0x02

    directory entry attribute bit: hidden
- #define AM_LFN 0x0F

    directory entry attribute bit: LFN entry
- #define AM_MASK 0x3F

    mask of defined directory entry attribute bits
- #define AM_RDO 0x01

    directory entry attribute bit: read only
- #define AM_SYS 0x04

    directory entry attribute bit: system
- #define AM_VOL 0x08

    directory entry attribute bit: Volume label
- #define div16SSZ(div)

    Divide (16 bit, unsigned) by fixed sector size.
- #define div2SSZ(div)

    Divide (32 bit, unsigned) by 2 times fixed sector size.
- #define div32SSZ(div)

    Divide (32 bit, unsigned) by fixed sector size.
- #define div4SSZ(div)

    Divide (32 bit, unsigned) by 4 times fixed sector size.
- #define divSSZ2(div)

    Divide (32 bit, unsigned) by fixed sector size by 2.
- #define divSSZ4(div)

    Divide (32 bit, unsigned) by fixed sector size by 4.
- #define f_eof(fp)

    Check EOF.
- #define f_error(fp)

    Check error state.
- #define f_size(fp)

    Get the size.
- #define f_tell(fp)

    Get the current read/write pointer.
- #define FA_CREATE_ALWAYS 0x08

    Mode flag: create anyway.
- #define FA_CREATE_NEW 0x04

    Mode flag: create a new file.
- #define FA_OPEN_ALWAYS 0x10

    Mode flag: open anyway.
- #define FA_OPEN_EXISTING 0

    Mode flag: open the existing file object.
- #define FA_READ 0x01

Mode flag: read access to the file object.
- #define FA_WRITE 0x02

  Mode flag: write access to the file object.
- #define FS_FAT12 1

  FATFS.fs_type value.
- #define FS_FAT16 2

  FATFS.fs_type value.
- #define FS_FAT32 3

  FATFS.fs_type value.
- #define get_fattime()

  Get current local FAT time.
- #define LD2PD(vol)

  Each logical drive is bound to the same physical drive number ∗/.
- #define LD2PT(vol)

  Always mounts the 1st partition or in SFD.
- #define modSSZ(div)

  Modulo (unsigned) by fixed sector size.
- #define mul16SSZ(fac)

  Multiply (16 bit, unsigned) with fixed sector size.
- #define mul32SSZ(fac)

  Multiply (32 bit, unsigned) with fixed sector size.
- #define SSM 511

  Mask for fixed sector size.
- #define SSZ 512

  Fixed sector size.

Enumerations

- enum FRESULT {
  FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY,
  FR_NO_FILE, FR_NO_PATH, FR_INVALID_NAME, FR_DENIED,
  FR_EXIST, FR_INVALID_OBJECT, FR_WRITE_PROTECTED, FR_INVALID_DRIVE,
  FR_NOT_ENABLED, FR_NO_FILESYSTEM, FR_MKFS_ABORTED, FR_TIMEOUT,
  FR_LOCKED, FR_NOT_ENOUGH_CORE, FR_TOO_MANY_OPEN_FILES, FR_INVALID_PARAMETER
  }

  File functions return code.

Functions

- FRESULT checkValidFS (uint8_t vol)

  Check if there's a valid mounted file system on the drive.
- FRESULT checkWriteProtect (uint8_t vol)

  Check if the drive is write protected.
- uint8_t extractDrive (const TCHAR ∗∗path)

  Extract drive number from path.
- FRESULT f_chdir (const TCHAR ∗)

  Change current directory (not implemented)
- FRESULT f_chdrive (uint8_t)

  Change current drive (not implemented)
- FRESULT f_chmod (const TCHAR ∗path, uint8_t value, uint8_t mask)

  Change attribute of the file or directory.

- FRESULT f_close (FIL ∗fp)

  Close an open file object.
- FRESULT f_fdisk (uint8_t pdrv, const uint32_t szt[ ], void ∗work)

  Divide a physical drive into some partitions.
- FRESULT f_getcwd (TCHAR ∗, uint16_t)

  Get current directory (not implemented)
- FRESULT f_getfree (const uint8_t vol, uint32_t ∗nclst)

  Get the number of free clusters on the drive.
- TCHAR ∗ f_gets (TCHAR ∗, int, FIL ∗)

  Get a string from the file (not implemented)
- FRESULT f_lseek (FIL ∗fp, uint32_t ofs)

  Move file pointer of a file object, expand file size.
- FRESULT f_mkdir (const TCHAR ∗path)

  Create a new directory.
- FRESULT f_mount (uint8_t vol, FATFS ∗fs)

  Mount / unmount a logical drive.
- FRESULT f_open (FIL ∗fp, const TCHAR ∗path, uint8_t mode)

  Open or create a file.
- FRESULT f_opendir (DIR ∗dj, const TCHAR ∗path)

  Open an existing directory.
- int f_printf (FIL ∗, const TCHAR ∗,...)

  Put a formatted string to the file (not implemented)
- int f_putc (TCHAR, FIL ∗)

  Put a character to the file (not implemented)
- int f_puts (const TCHAR ∗, FIL ∗)

  Put a string to the file (not implemented)
- FRESULT f_read (FIL ∗fp, void ∗buff, uint16_t btr, uint16_t ∗br)

  Read data from a file.
- FRESULT f_readdir (DIR ∗dj, FILINFO ∗fno)

  Read a directory item.
- FRESULT f_rename (const TCHAR ∗path_old, const TCHAR ∗path_new)

  Rename or move a file or directory.
- FRESULT f_stat (const TCHAR ∗path, FILINFO ∗fno)

  Get file status.
- FRESULT f_sync (FIL ∗fp)

  Flush written / cached data to a file.
- FRESULT f_truncate (FIL ∗fp)

  Truncate file.
- FRESULT f_unlink (const TCHAR ∗path)

  Delete an existing file or directory.
- FRESULT f_utime (const TCHAR ∗path, const FILINFO ∗fno)

  Change time-stamps of a file or directory.
- FRESULT f_write (FIL ∗fp, const void ∗buff, uint16_t btw, uint16_t ∗bw)

  Write data to a file.
- FRESULT getValidFS (uint8_t vol, FATFS ∗∗fatfs)

  Get the file system for the drive.
- FRESULT lookForFS (uint8_t vol)

  Look for a recognised file system on the drive.

4.11 include/fatFS/ffconf.h File Reference

4.11.1 Overview

(Grande) fatFS configuration options This is the include file for ChaN's (grande) fatFS for small memory cards (SMC, MMC) and FAT file system functions. It contains basic option settings.

Copyright (c) 2012, ChaN, all right reserved.

Modification for weAutSys by A. Weinert

Copyright © 2012 Albrecht Weinert

weinert - automation, Bochum

weinert-automation.de weAut.de a-weinert.de

Look at file fatFS/ff.h for details.

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Defines

- #define _CODE_PAGE 1

  The code page used.
- #define _FS_MINIMIZE 0

  Omit functions to save (program / flash) memory.
- #define _FS_READONLY 0

  Only read functions.
- #define _FS_RPATH 0

  The relative path feature.
- #define _LFN_UNICODE 0

  0: ANSI/OEM or 1: Unicode
- #define _MAX_LFN 255

  Maximum LFN length to handle (12 to 255)
- #define _MULTI_PARTITION 0

  Allow multiple partitions.
- #define _USE_ERASE 0

  Allow erase of blocks of sectors.
- #define _USE_FASTSEEK 0

  0: disable or 1: enable the fast seek feature
- #define _USE_LFN 0

  The long file name feature.
- #define _USE_STRFUNC 0

  0: disable or 1-2: enable string functions
- #define _VOLUMES 1

  Number of volumes (logical drives) to be used.
- #define _WORD_ACCESS 1

  Word access.

4.12 include/uip/apps/dhcpc.h File Reference

4.12.1 Overview

Definitions for the DHCP client. The Dynamic Host Configuration Protocol's (DHCP) purpose is getting IP configuration data from a server.

Adam Dunkels' original copyright notice

Copyright (c) 2005, Swedish Institute of Computer Science. All rights reserved.

(contained in this file) still holds. For modifications:

Copyright © 2011 Albrecht Weinert, Bochum

Author

Adam Dunkels `adam@dunkels.com`
Albrecht Weinert <`a-weinert.de`>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Data Structures

- struct dhcpMsg_t

  The DHCP message structure.
- struct dhcpOption_t

  The structure of a DHCP message option field.
- struct dhcpState_t

  Structure for DHCP application state.

Defines

- #define BOOTP_BROADCAST 0x8000

  DHCP message flag value.
- #define DHCP_HLEN_ETHERNET 6

  DHCP message hardware address length.
- #define DHCP_HTYPE_ETHERNET 1

  DHCP message type value.
- #define DHCP_OPTION_BROADCAST_ADDR 28

  DHCP option field code: broadcast address.
- #define DHCP_OPTION_CLIENT_NAME 12

  DHCP option field code: host name.
- #define DHCP_OPTION_DEFAULT_TTL 23

  DHCP option field code: default IP TTL.
- #define DHCP_OPTION_DNS_SERVER 6

  DHCP option field code.
- #define DHCP_OPTION_DOMAIN_NAME 15

  DHCP option field code: domain name.
- #define DHCP_OPTION_END 255

DHCP option field code: end of all options.

- #define DHCP_OPTION_LEASE_TIME 51

  The address lease time.
- #define DHCP_OPTION_MSG_TYPE 53

  DHCP option field code: message type.
- #define DHCP_OPTION_NAME_SERVERS 5

  DHCP option field code: name server(s)
- #define DHCP_OPTION_NTP_SERVERS

  DHCP option field code: time server(s)
- #define DHCP_OPTION_REBND_TIME 59

  DHCP option field code.
- #define DHCP_OPTION_RENEW_TIME 58

  DHCP option field code.
- #define DHCP_OPTION_REQ_IPADDR 50

  DHCP option field code.
- #define DHCP_OPTION_REQ_LIST 55

  DHCP request option field code: Parameter Request List.
- #define DHCP_OPTION_ROUTER 3

  DHCP option field code.
- #define DHCP_OPTION_SERVER_ID 54

  DHCP option field code: DHCH server IP address.
- #define DHCP_OPTION_SMTP_SERVERS 69

  DHCP option field code: SMTP server(s)
- #define DHCP_OPTION_SUBNET_MASK 1

  DHCP option field code.
- #define DHCP_OPTION_TIME_OFFSET 2

  DHCP option field code: time offset.
- #define DHCP_OPTION_TIME_SERVERS 4

  DHCP option field code: time server(s)
- #define DHCP_REPLY 2

  DHCP message operation type.
- #define DHCP_REQUEST 1

  DHCP message operation type.
- #define DHCPACK 5

  DHCP message type.
- #define DHCPC_CLIENT_PORT 68

  Used DHCP client port.
- #define DHCPC_SERVER_PORT 67

  Well known DHCP server port.
- #define DHCPDECLINE 4

  DHCP message type.
- #define DHCPDISCOVER 1

  DHCP message type.
- #define DHCPINFORM 8

  DHCP message type.
- #define DHCPNAK 6

  DHCP message type.
- #define DHCPOFFER 2

  DHCP message type.
- #define DHCPRELEASE 7

  DHCP message type.

- #define DHCPREQUEST 3

    DHCP message type.
- #define STATE_CONFIG_RECEIVED 3

    DHCP state machine state.
- #define STATE_INITIAL 0

    DHCP state machine state.
- #define STATE_OFFER_RECEIVED 2

    DHCP state machine state.
- #define STATE_SENDING 1

    DHCP state machine state.

Functions

- ptfnct_t dhcpc_appcall (void)

    Handle DHCP server events.
- void dhcpc_configured (uint8_t respType, const uint16_t ipAddr[ ])

    DHCP success.
- void dhcpcGotOption (const struct dhcpOption_t ∗dhcpOption)

    DHCP option received.
- void dhcpInit (void const ∗mac_addr)

    Initialise the DHCP (client)
- void dhcpReset (void)

    Reset the DHCP (client)

Variables

- struct dhcpState_t dhcpState

    DHCP application state.

4.13 include/uip/apps/resolv.h File Reference

4.13.1 Overview

DNS resolver definitions. Adam Dunkels' original copyright notice

Copyright (c) 2002-2003, Adam Dunkels. All rights reserved.

(contained in this file) still holds. For modifications:

Copyright © 2011 Albrecht Weinert, Bochum

Author

Adam Dunkels adam@dunkels.com
Albrecht Weinert <a-weinert.de>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Defines

- #define MAX_DNS_RETRIES 8

    The maximum number of DNS retries.

Functions

- void resolv_appcall (void)

    uIP event function for the DNS resolver
- void resolv_conf (uint16_t ∗dnsserver)

    Configure which DNS server to use for queries.
- void resolv_found (char ∗name, uint16_t ∗ipaddr)

    Callback function which is called after hostname look-up.
- uint16_t ∗ resolv_getserver (void)

    Obtain the currently configured DNS server.
- void resolv_init (void)

    Initialise the resolver.
- uint16_t ∗ resolv_lookup (char ∗name)

    Look up a hostname in the array of known hostnames.
- void resolv_query (char ∗name)

    Queues a name so that a question for the name will be sent out.

4.14 include/uip/uip.h File Reference

4.14.1 Overview

Common definitions for the uIP TCP/IP stack.

Author

Adam Dunkels `adam@dunkels.com`

The uIP TCP/IP stack header file contains definitions for a number of C macros that are used by uIP programs as well as internal uIP structures, TCP/IP header structures and function declarations.

Data Structures

- struct uip_eth_addr

    Representation of a 48-bit Ethernet address / MAC address.
- struct uip_icmpip_hdr

    The ICMP and IP headers.
- struct uip_tcpip_hdr

    The TCP and IP headers.
- struct uip_udpip_hdr

    The UDP and IP headers.
- struct uipConn_t

    Representation of a uIP TCP connection.

Defines

- #define HTONS(n)

  Convert 16-bit quantity from host byte order to network byte order.
- #define uip_abort()

  Abort the current connection.
- #define uip_aborted()

  Has the connection been aborted by the other end.
- #define uip_acked()

  Has previously sent data been acknowledged?
- #define UIP_APPDATA_SIZE

  The buffer size available for user data in the buffer uip_buf.
- #define uip_close()

  Close the current connection.
- #define uip_closed()

  Has the connection been closed by the other end?
- #define UIP_CLOSED

  state value in uip_conn->tcpstateflags
- #define UIP_CLOSING

  state value in uip_conn->tcpstateflags
- #define uip_conn_active(conn)

  Check if a connection identified by its number is active.
- #define uip_connected()

  Has the connection just been connected?
- #define UIP_DATA

  flag to uip_process(): incoming data in uip_buf
- #define uip_datalen()

  The length of any incoming data that is currently available (if available) in the uip_appdata buffer.
- #define UIP_ESTABLISHED

  state value in uip_conn->tcpstateflags
- #define UIP_FIN_WAIT_1

  state value in uip_conn->tcpstateflags
- #define UIP_FIN_WAIT_2

  state value in uip_conn->tcpstateflags
- #define uip_getdraddr(addr)

  Get the default router's IP address.
- #define uip_gethostaddr(addr)

  Get the IP address of this host.
- #define uip_getnetmask(addr)

  Get the netmask.
- #define uip_initialmss()

  Get the initial maximum segment size (MSS) of the current connection.
- #define uip_input()

  Process an incoming packet.
- #define uip_ip6addr(addr, addr0, addr1, addr2, addr3, addr4, addr5, addr6, addr7)

  Construct an IPv6 address from eight 16-bit words.
- #define uip_ipaddr(addr, addr0, addr1, addr2, addr3)

  Construct an IP address from four bytes.
- #define uip_ipaddr1(addr)

  Pick the first octet of an IP address.
- #define uip_ipaddr2(addr)

Pick the second octet of an IP address.
- #define uip_ipaddr3(addr)

    Pick the third octet of an IP address.
- #define uip_ipaddr4(addr)

    Pick the fourth octet of an IP address.
- #define uip_ipaddr_cmp(addr1, addr2)

    Compare two IP addresses.
- #define uip_ipaddr_copy(dest, src)

    Copy an IP address to another IP address.
- #define uip_ipaddr_mask(dest, src, mask)

    Mask out the network part of an IP address.
- #define uip_ipaddr_maskcmp(addr1, addr2, mask)

    Compare two IP addresses with netmasks.
- #define UIP_LAST_ACK

    state value in uip_conn->tcpstateflags
- #define uip_mss()

    Current maximum segment size that can be sent on the current connection.
- #define uip_newdata()

    Is new incoming data available?
- #define uip_periodic(conn)

    Periodic processing for a connection identified by its number.
- #define uip_periodic_conn(conn)

    Perform periodic processing for a connection identified by pointer.
- #define uip_poll()

    Is the connection being polled by uIP?
- #define uip_poll_conn(conn)

    Request that a particular connection should be polled.
- #define UIP_POLL_REQUEST

    flag to uip_process(): poll a connection
- #define uip_restart()

    Restart the current connection, if is has previously been stopped with uip_stop()
- #define uip_rexmit()

    Do we need to retransmit previously data?
- #define uip_setdraddr(addr)

    Set the default router's IP address.
- #define uip_sethostaddr(addr)

    Set the IP address of this host.
- #define uip_setnetmask(addr)

    Set the netmask.
- #define uip_stop()

    Tell the sending host to stop sending data.
- #define uip_stopped(conn)

    Find out if the current connection has been previously stopped with uip_stop()
- #define UIP_STOPPED

    extra state bit in uip_conn->tcpstateflags
- #define UIP_SYN_RCVD

    state value in uip_conn->tcpstateflags
- #define UIP_SYN_SENT

    state value in uip_conn->tcpstateflags
- #define UIP_TIME_WAIT

    state value in uip_conn->tcpstateflags

- #define uip_timedout()

  Has the connection timed out?
- #define UIP_TIMER

  flag to uip_process(): periodic timer has fired
- #define UIP_TS_MASK

  mask for state values in uip_conn->tcpstateflags
- #define uip_udp_bind(conn, port)

  Bind a UDP connection to a local port.
- #define uip_udp_remove(conn)

  Remove / give up an UDP connection.
- #define uip_udp_send(len)

  Send a UDP datagram of length len on the current connection.
- #define UIP_UDP_SEND_CONN

  flag to uip_process(): construct UDP datagram in uip_buf
- #define uip_udpconnection()

  Is the current connection a UDP connection?

Typedefs

- typedef uint16_t uip_ip4addr_t [2]

  Representation of an IP V4 address.
- typedef uint16_t uip_ip6addr_t [8]

  Representation of an IP (V6) address.
- typedef uip_ip4addr_t uip_ipaddr_t

  IP address is of type IP V4.

Functions

- uint16_t chksum (uint16_t sum, const uint8_t ∗data, uint16_t len)

  Calculate the Internet checksum over a buffer.
- uint16_t uip_chksum (uint16_t ∗buf, uint16_t len)

  Calculate the Internet checksum over a buffer.
- struct uipConn_t ∗ uip_connect (uip_ipaddr_t ∗ripaddr, uint16_t port)

  Connect to a remote host using TCP.
- void uip_init (void)

  uIP initialisation function
- uint16_t uip_ipchksum (void)

  Calculate the IP header checksum of the packet header in uip_buf.
- void uip_listen (uint16_t port)

  Start listening on the specified port.
- void uip_process (uint8_t flag)

  The actual uIP function which does all the work.
- void uip_send (const void ∗data, int len)

  Send data on the current connection.
- void uip_setipid (uint16_t id)

  uIP initialisation function
- uint16_t uip_tcpchksum (void)

  Calculate the TCP checksum of the packet in uip_buf and uip_appdata.
- struct uip_udp_conn ∗ uip_udp_new (uip_ipaddr_t ∗ripaddr, uint16_t rport)

  Set up a new UDP connection.

- uint16_t uip_udpchksum (void)

  Calculate the UDP checksum of the packet in uip_buf and uip_appdata.
- void uip_unlisten (uint16_t port)

  Stop listening on the specified port.
- void uipSecTick (void)

  One second tick for the uiP stack.

Variables

- struct uip_eth_addr actMACadd

  The (actual) MAC address.
- void ∗ uip_appdata

  Pointer to the application data in the packet buffer.
- uint8_t uip_buf []

  The uIP packet buffer.
- char ∗ uip_buf_alias

  alias to uip_buf
- struct uipConn_t ∗ uip_conn

  Pointer to the current TCP connection.
- struct uipConn_t uip_conns []

  The array containing all uIP connections.
- uint16_t uip_len

  The length of the packet in the uip_buf buffer.

4.15 include/uip/uip_arp.h File Reference

4.15.1 Overview

Definitions for the ARP module. Adam Dunkels' original copyright notice

Copyright (c) 2001-2003, Adam Dunkels. All rights reserved.

(contained in this file) still holds. For modifications:

Copyright © 2011 Albrecht Weinert, Bochum

Author

Adam Dunkels adam@dunkels.com
Albrecht Weinert <a-weinert.de>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Data Structures

- struct arp_entry

  An ARP entry.
- struct arp_hdr

The ARP header.

- struct ethip_hdr

  The IP header.

- struct uip_eth_hdr

  The Ethernet header.

Defines

- #define ARP_HWTYPE_ETH

  ARP packet hardware Ethernet (the only supported one)

- #define ARP_REPLY

  packet type answer "I have IP.. and MAC.."

- #define ARP_REQUEST

  packet type request "which MAC has IP...?"

- #define UIP_ETHTYPE_ARP

  Type ARP (in little endian)

- #define UIP_ETHTYPE_IP

  Type IP (V.4) (in little endian)

- #define UIP_ETHTYPE_IP6

  Type IPV6 (in little endian)

Functions

- void uip_arp_arpin (void)

  ARP processing for incoming ARP packets.

- void uip_arp_init (void)

  Initialise the ARP module.

- void uip_arp_ipin (void)

  ARP processing for incoming IP packets.

- void uip_arp_out (void)

  The ARP output prepare function.

- void uip_arp_timer (void)

  Periodic ARP processing function.

Variables

- struct arp_entry arp_table [ ]

  The ARP table.

- struct uip_eth_addr broadcast_ethaddr

  The broadcast MAC.

- const uint16_t broadcast_ipaddr [ ]

  The broadcast IP address.

4.16 include/uip/uipopt.h File Reference

4.16.1 Overview

Configuration options for uIP. This file is used for tweaking various configuration options for uIP. You should make a copy of this file into one of your project's directories instead of editing this example "uipopt.h" file that comes with the uIP distribution.

This file is part of the uIP TCP/IP stack.

Modification for weAutSys by A. Weinert concern: typos, Doxygen errors and adaption.

Adam Dunkels' original copyright notice

Copyright (c) 2001-2003, Adam Dunkels. All rights reserved.

(contained in this file) still holds. For modifications:

Copyright © 2011 Albrecht Weinert, Bochum

Author

Adam Dunkels `adam@dunkels.com`
Albrecht Weinert <`a-weinert.de`>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Defines

Static configuration options

These configuration options can be used for setting the IP address settings statically, but only if UIP_FIXEDADDR is set to 1.

The configuration options for a specific node includes IP address, netmask and default router as well as the Ethernet address. The netmask, default router and Ethernet address are applicable only if uIP should be run over Ethernet.

All of these should be changed to suit your project.

- #define UIP_FIXEDADDR
  Determines if uIP should use a fixed IP address or not.
- #define UIP_PINGADDRCONF
  Ping IP address assignment.

IP configuration options

- #define UIP_TTL
  The IP TTL (time to live) of IP packets sent by uIP.
- #define UIP_REASSEMBLY
  Turn on support for IP packet re-assembly.
- #define UIP_REASS_MAXAGE
  maximum wait time an IP fragment in the re-assembly buffer

UDP configuration options

- #define UIP_UDP
  UDP support should be compiled in (or not)

- #define UIP_UDP_CHECKSUMS

  Checksums should be used (or not)
- #define UIP_INC_CHECKSUMS 0

  Check incoming checksums on or off.
- #define UIP_UDP_CONNS

  Maximum amount of concurrent UDP connections.

TCP configuration options

- #define UIP_ACTIVE_OPEN

  Support for opening connections from uIP should be compiled in (ot not)
- #define UIP_CONNS

  The maximum number of simultaneously open TCP connections.
- #define UIP_LISTENPORTS

  The maximum number of simultaneously listening TCP ports.
- #define UIP_URGDATA

  Support for TCP urgent data notification should be compiled in (or not)
- #define UIP_RTO

  The initial retransmission timeout counted in timer pulses.
- #define UIP_MAXRTX

  The maximum number of times a segment should be retransmitted.
- #define UIP_MAXSYNRTX

  The maximum number of times a SYN segment should be retransmitted.
- #define UIP_TCP_MSS

  The TCP maximum segment size.
- #define UIP_RECEIVE_WINDOW

  The size of the advertised receiver's window.
- #define UIP_TIME_WAIT_TIMEOUT

  How long a connection should stay in the TIME_WAIT state.

ARP configuration options

- #define UIP_ARPTAB_SIZE

  The size of the ARP table.
- #define UIP_ARP_MAXAGE

  The maximum age of ARP table entries measured in 10 seconds unit.

CPU architecture configuration

The CPU architecture configuration is where the endianess of the CPU on which uIP is to be run is specified.

Most CPUs today are little endian, including Intel (x86) and ATmega. The most notable exception are Motorola's CPUs which are big endian. The BYTE_ORDER macro must reflect the CPU architecture on which uIP is to be run.

- #define UIP_BYTE_ORDER

  The byte order of the CPU on which uIP is to be run.
- #define UIP_ARCH_ADD32

  There is a platform implementation of 32bit big endian addition.

General configuration options

- #define UIP_BUFSIZE

  The size of the uIP packet buffer.
- #define UIP_STATISTICS

  Statistics support should be compiled in (or not)
- #define UIP_LOGGING

  Logging of certain events should be compiled in (or not)

- #define UIP_BROADCAST

  Broadcast support.
- #define UIP_LLH_LEN

  The link level header length.
- void uip_log (char ∗msg)

  Print out a uIP log message.

Typedefs

Application specific configurations

An uIP application is implemented using a single application function that is called by uIP whenever a TCP/IP event occurs.

The name of this function must be registered with uIP at compile time using the UIP_APPCALL definition.

uIP applications can store the application state within the uip_conn structure by specifying the type of the application structure by typedef'ing the type uip_tcp_appstate_t and uip_udp_appstate_t.

The file containing the definitions must be included in the uipopt.h file.
The following example illustrates how this can look.

```
void httpd_appcall(void);
#define UIP_APPCALL     httpd_appcall

struct httpd_state {
  uint8_t state;
  uint16_t count;
  char *dataptr;
  char *script;
};
typedef struct httpd_state uip_tcp_appstate_t
```

- typedef uint16_t uip_udp_appstate_t

  The type of appstate in an uip_udp_conn.

4.17 include/we-aut_sys/cli.h File Reference

4.17.1 Overview

weAutSys' command line interpreter (CLI) This file contains the definitions for weAutSys' system threads and functions for the command line interpretation. Please find detailed description of the variables, functions etc. in the module Command line interpreter and also at system threads.

This is system software and must not be modified for user or application programs.

This file is part of weAutSys <weinert-automation.de>

Copyright © 2014 Albrecht Weinert, Bochum

Author

Albrecht Weinert <a-weinert.de>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Data Structures

- struct appCLIreg_t

  The user / application CLI registration type.

Defines

- #define YIELD_FOR_BUSY_CLI(appThread)

  Wait (yielding) for command execution to end.

Functions

- ptfnct_t appCliThreadF (struct cliThr_data_t ∗cliThread)

  The user / application specific command line interpreter thread.
- void initAsCLIthread (struct thr_data_t ∗thread, FILE ∗file)

  Initialise a thread (structure) as command line interpreter (CLI) thread.
- void registerAppCli (p2ptFunA threadF, char const ∗const userCommands[ ])

  Register a user / application command line interpreter (CLI)
- uint8_t setCliLine (struct cliThr_data_t ∗const cliThread, char ∗const line, uint16_t length)

  Set a command line in the command line interpreter (CLI) thread data.
- ptfnct_t sysCliThreadF (struct cliThr_data_t ∗cliThread)

  The system command line interpreter thread.
- void unimplOptionReport (struct cliThr_data_t ∗cliThread)

  Report the use of an un-implemented option for a command.

Variables

- struct appCLIreg_t appCLIreg

  The user / application command line interpreter (CLI) registration.

4.18 include/we-aut_sys/common.h File Reference

4.18.1 Overview

weAutSys' common types and helper functions This file contains the definitions for weAutSys' common types and helper macros or functions. They are utilised in more than one application field respectively application or system module &mdash but not in bootloaders.

This file is part of weAutSys <`weinert-automation.de`>

Copyright © 2012 Albrecht Weinert, Bochum

Author

Albrecht Weinert <`a-weinert.de`>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Data Structures

- struct cliThr_data_t

  The organisational data for a command line interpreter (CLI) thread.
- struct hierThr_data_t

  The organisational data for a small thread hierarchy.
- struct modConfData_t

  The configuration data (type) for a Modbus handling.
- struct modTelegr_t

  The (start of a) Modbus TCP/IP telegram.
- struct modThr_data_t

  The organisational data for a Modbus handler thread.
- struct mThr_data_t

  State data for a thread (minimal)
- struct outFlashTextThr_data_t

  The organisational data for a flash strings array output thread.
- struct thr_data_t

  State data for a thread (universal variable type)

Defines

- #define EXTRA_THR_ST_SZ 158

  Size of (extra) thread state data.
- #define FOLLOW_UP

  Follow up string marker for outFlashTextThr_data_t.
- #define LEN_OF_CLITHR_LINE

  The maximum number of characters in cliThr_data_t.line.
- #define SIZE_OF_BIGGEST_APPSTATE 164

  The size of the biggest application state structure used in bytes.

Typedefs

- typedef ptfnct_t(fun_t)(struct thr_data_t ∗thrData)

  Type of a protothread function (struct thr_data_t ∗)
- typedef ptfnct_t(funA_t)(struct cliThr_data_t ∗thrData)

  Type of a protothread function (cliThr_data_t ∗)
- typedef ptfnct_t(funM_t)(struct modThr_data_t ∗thrData)

  Type of a protothread function (modThr_data_t ∗)
- typedef ptfnct_t(funU_t)(struct mThr_data_t ∗uthr_data)

  Type of a protothread function (struct mThr_data_t ∗)
- typedef ptfnct_t(funV_t)(void)

  Type of a protothread function (void)
- typedef funU_t ∗ p2ptFun

  Pointer to a protothread function (struct mThr_data_t ∗)
- typedef funA_t ∗ p2ptFunA

  Pointer to a protothread function (cliThr_data_t ∗)
- typedef fun_t ∗ p2ptFunC

  Pointer to a protothread function (struct thr_data_t ∗)
- typedef funM_t ∗ p2ptFunM

  Pointer to a protothread function (modThr_data_t ∗)
- typedef funV_t ∗ p2ptFunV

  Pointer to a protothread function (void)

Variables

- char const bLF1 []

  1 blank, 1 linefeed 0 terminated in flash memory
- char const bLF2 []

  1 blank, 2 linefeed 0 terminated in flash memory
- char const l4nefeeds []

  A short string with just one blank and four linefeeds in flash memory.

4.19 include/we-aut_sys/enc28j60.h File Reference

4.19.1 Overview

weAutSys' (weAut_01's) 28J60 Ethernet driver This file contains the definitions of (low level) system calls and services for the weAutSys' (weAut_01's) 28J60 Ethernet driver.

This is system software and must not be modified for user or application programs.

This file is part of weAutSys <`weinert-automation.de`>

Copyright © 2011 Albrecht Weinert, Bochum

Author

Albrecht Weinert <`a-weinert.de`>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Defines

- #define BFC_CMD

  Bit field clear command.
- #define BFS_CMD

  Bit field set command.
- #define COLSTAT

  collision is occurring (Bit number)
- #define deSelectEnc()

  De-select the ENC28J60.
- #define DPXSTAT

  configured for full-duplex (Bit number)
- #define ENCBUF_END 0x1FFF

  ENC's dual port RAM end.
- #define FRCLNK

  make link up even when no partner station detected (Bit number)
- #define HDLDIS

  half duplex loopback disable (Bit number)
- #define JABBER

  turn jabber correction off (Bit number)
- #define JBSTAT

transmission met jabber criteria since last PHSTAT1 read
- #define LA_BFAST

  see PHLCON
- #define LA_BSLOW

  see PHLCON
- #define LA_COL

  see PHLCON
- #define LA_DSC
- #define LA_DSTAT

  see PHLCON
- #define LA_LEDOFF

  see PHLCON
- #define LA_LEDON

  see PHLCON
- #define LA_LSRX

  see PHLCON
- #define LA_LSTAT

  see PHLCON
- #define LA_LSTXRX

  see PHLCON
- #define LA_RX

  see PHLCON
- #define LA_RXTX

  see PHLCON
- #define LA_TX

  see PHLCON
- #define LB_BFAST

  see PHLCON
- #define LB_BSLOW

  see PHLCON
- #define LB_COL

  see PHLCON
- #define LB_DSC

  see PHLCON
- #define LB_DSTAT

  see PHLCON
- #define LB_LEDOFF

  see PHLCON
- #define LB_LEDON

  see PHLCON
- #define LB_LSRX

  see PHLCON
- #define LB_LSTAT

  see PHLCON
- #define LB_LSTXRX

  see PHLCON
- #define LB_RX

  see PHLCON
- #define LB_RXTX

  see PHLCON
- #define LB_TX

see PHLCON
- #define LLSTAT

  link was up continuously since last PHSTAT1 read
- #define LSTAT

  link is currently up (Bit number)
- #define NOSTRCH

  do not stretch LED events (see PHLCON)
- #define PFDPX

  full duplex capable (bit 12 is always set)
- #define PHCON1

  PHY control register 1.
- #define PHCON2

  PHY control register 2.
- #define PHDPX

  half duplex capable (bit 11 is always set)
- #define PHID1

  PHY identification register 1.
- #define PHID2

  PHY identification register 2.
- #define PHIE

  PHY interrupt enable register.
- #define PHIR

  PHY interrupt request register.
- #define PHLCON

  LED Configuration Register.
- #define PHSTAT1

  PHY status register 1.
- #define PHSTAT2

  PHY status register 2.
- #define PLRITY

  polarity of TPIN is reversed
- #define RBM_CMD

  Read buffer memory command.
- #define RCR_CMD

  Read control register command.
- #define RXBUF_END 0x1A0D

  Receive buffer end.
- #define RXSTAT

  PHY is currently receiving (Bit number)
- #define selectEnc()

  Chip select the ENC28J60.
- #define SYSRST_CMD 0xFF

  Reset ENC28J60 command.
- #define TLSTRCH

  LED pulse stretch, long 140ms (see PHLCON)
- #define TMSTRCH

  LED pulse stretch, medium 70ms (see PHLCON)
- #define TNSTRCH

  LED pulse stretch, normal 40ms (see PHLCON)
- #define tranceiveEnc(sendB)

  Transmit and receive one byte to / from ENC28J60.

- #define tranceiveEnc2(sendB1, sendB2)

    Transmit two bytes and receive one byte to / from ENC28J60.
- #define TXBUF_STRT 0x1A0E

    Transmit buffer start.
- #define TXDIS

    disable twisted-pair transmitter hardware driver (Bit number)
- #define TXSTAT

    PHY is currently transmitting (Bit number)
- #define WBM_CMD

    Write buffer memory command.
- #define WCR_CMD

    Write control register command.

Control registers

- #define ESTAT

    ETHERNET status register.
- #define CLKRDY

    Clock (oscillator) is ready (ESTAT bit number)
- #define TXABRT

    The transmit request was aborted (ESTAT bit number)
- #define ECON2

    ETHERNET control register 2.
- #define AUTOINC

    automatic increment and wrap of RAM buffer addresses (ECON2 bit number)
- #define PKTDEC

    decrement (received) packets count (ECON2 bit number)
- #define PWRSV

    switch the PHY interface off (save power when not needed; ECON2 bit number)
- #define VRPS

    only if PWRSV set voltage regulator to low current (ECON2 bit number)
- #define ECON1

    ETHERNET control register 1.
- #define TXRST

    Transmit only reset (ECON1 bit number)
- #define RXRST

    Receive only reset (ECON1 bit number)
- #define DMAST

    DMA operation (within internal RAM) start and busy bit (ECON1 bit number)
- #define CSUMEN

    DMA hardware calculates checksums (ECON1 bit number)
- #define TXRTS

    The transmit logic is attempting to transmit a packet (ECON1 bit number)
- #define RXEN

    Receive enable.
- #define EIE

    Ethernet interrupt enable register.
- #define INTIE

    EIE bit number
- #define PKTIE

    EIE bit number
- #define DMAIE

    EIE bit number
- #define LINKIE

EIE bit number
- #define TXIE

  EIE bit number
- #define TXERIE

  EIE bit number
- #define RXERIE

  EIE bit number
- #define EIR

  Ethernet interrupt request register.
- #define PKTIF

  EIR bit number
- #define DMAIF

  EIR bit number
- #define LINKIF

  EIR bit number
- #define TXIF

  EIR bit number
- #define TXERIF

  EIR bit number
- #define RXERIF

  EIR bit number

Per packet control bits for transmission settings

- #define PHUGEEN

  Huge frame enable (bit number)
- #define PPADEN

  per packet padding enable (bit number)
- #define PCRCEN

  per packet CRC enable (bit number)
- #define POVERRIDE

  packet overrides (bit number)

Symbolic names for direct register access

The lower 5 bits (0..31) give the register number in each bank.

The bits 5 and 5 (mask 0x60) are used for the bank number (0..3 $*$ 32).

- #define ERDPTL

  The buffer read address register.
- #define ERDPTH

  the buffer read address register
- #define EWRPTL

  The buffer write address register.
- #define EWRPTH

  the buffer write address register
- #define ETXSTL

  The write / transmit buffer range.
- #define ETXSTH

  the write buffer range
- #define ETXNDL

  the write buffer range
- #define ETXNDH

the write buffer range

- #define ERXSTL

  The read / receive buffer range.
- #define ERXSTH

  the write buffer range
- #define ERXNDL

  the write buffer range
- #define ERXNDH

  the write buffer range
- #define ERXRDPTL

  The receive buffer (already) read address register.
- #define ERXRDPTH

  receive buffer already read
- #define ERXWRPTL

  The receive buffer write address register.
- #define ERXWRPTH

  receive buffer write
- #define EDMASTL

  DMA start low byte.
- #define EDMASTH

  DMA start high byte.
- #define EDMANDL

  DMA end low byte.
- #define EDMANDH

  DMA end high byte.
- #define EDMADSTL

  DMA destination low byte.
- #define EDMADSTH

  DMA destination high byte.
- #define EDMACSL

  DMA checksum low byte.
- #define EDMACSH

  DMA checksum high byte.
- #define EHT0

  hash table byte 0
- #define EHT1

  hash table byte 1
- #define EHT2

  hash table byte 2
- #define EHT3

  hash table byte 3
- #define EHT4

  hash table byte 4
- #define EHT5

  hash table byte 5
- #define EHT6

  hash table byte 6
- #define EHT7

  hash table byte 7
- #define EPMM0

  pattern match mask byte 0

- #define EPMM1

  pattern match mask byte 1
- #define EPMM2

  pattern match mask byte 2
- #define EPMM3

  pattern match mask byte 3
- #define EPMM4

  pattern match mask byte 4
- #define EPMM5

  pattern match mask byte 5
- #define EPMM6

  pattern match mask byte 6
- #define EPMM7

  pattern match mask byte 7
- #define EPMCSL

  pattern match checksum low byte
- #define EPMCSH

  pattern match checksum high byte
- #define EPMOL

  pattern match offset low byte
- #define EPMOH

  pattern match offset high byte
- #define ERXFCON

  Ethernet receive filter control register.
- #define EPKTCNT

  Ethernet package count.
- #define MACON1

  MAC control register 1.
- #define TXPAUS

  MAC control register 1 (bit number)
- #define RXPAUS

  MAC control register 1 (bit number)
- #define PASSALL

  MAC control register 1 (bit number)
- #define MARXEN

  MAC control register 1 (bit number)
- #define MACON2

  MAC control register 2 (inofficial)
- #define MACON3

  MAC control register 3.
- #define PADCFG2

  MAC control register 3 (bit number)
- #define PADCFG1

  MAC control register 3 (bit number)
- #define PADCFG0

  MAC control register 3 (bit number)
- #define EXPSF64

  All short frames will be padded to 64 Bytes and have valid CRC appended.
- #define EXPSF60

  All short frames will be padded to 60 Bytes and have valid CRC appended.
- #define NOSFPAD

No handling short frames.
- #define DTCTVLAN

    Automatic handling short frames.
- #define TXCRCEN

    MAC control register 3 (bit number)
- #define PHIDREN

    MAC control register 3 (bit number)
- #define HFRMEN

    MAC control register 3 (bit number)
- #define FRMLNEN

    MAC control register 3 (bit number)
- #define FULDPX

    MAC control register 3 (bit number)
- #define MACON4

    MAC control register 4.
- #define DEFER

    MAC control register 4 (bit number)
- #define BPEN

    MAC control register 4 (bit number)
- #define NOBKOFF

    MAC control register 4 (bit number)
- #define MABBIPG

    back-to-back inter-packet gap
- #define MAIPGL

    non back-to-back inter-packet gap low
- #define MAIPGH

    non back-to-back inter-packet gap high
- #define MACLCON1

    retransmission maximum (4 bit)
- #define MACLCON2

    collision window (6 bit)
- #define MAMXFLL

    maximum frame length low byte
- #define MAMXFLH

    maximum frame length high byte
- #define MICMD

    MII command register.
- #define MIISCAN

    MII command register (bit number)
- #define MIIRD

    MII command register (bit number)
- #define MIREGADR

    MII register address register.
- #define MIWRL

    MII write data low byte.
- #define MIWRH

    MII write data high byte.
- #define MIRDL

    MII read data low byte.
- #define MIRDH

    MII read data high byte.

- #define MAADR1

  MAC address register (first)
- #define MAADR2

  MAC address register.
- #define MAADR3

  MAC address register.
- #define MAADR4

  MAC address register.
- #define MAADR5

  MAC address register.
- #define MAADR6

  MAC address register (last)
- #define EBSTSD

  built-in self test fill seed
- #define EBSTCON

  Ethernet self test control register.
- #define EBSTCSL

  built-in self test checksum low byte
- #define EBSTCSH

  built-in self test checksum high byte
- #define MISTAT

  MII status register.
- #define NVALID

  MII status register (bit number)
- #define SCAN

  MII status register (bit number)
- #define BUSY

  MII status register (bit number)
- #define EREVID

  Ethernet revision ID (5 bit, R/O)
- #define ECOCON

  Clock output control.
- #define EFLOCON

  EFLOCON (Ethernet Flow Control.
- #define EPAUSL

  Pause timer value low byte.
- #define EPAUSH

  Pause timer value high byte.
- #define REG_MASK 0x1F

  Mask for register number bits in symbolic register address.
- #define REG_BANK_MASK 0x60

  Mask for bank bits in symbolic register address.
- #define SPI_BANK_MASK

  Mask for bank bits in control register.
- #define KEY_REGISTERS 0x1B

  First number of common registers.
- uint8_t currentBank

  The currently set bank of registers.

Functions

- void clearBitfield (uint8_t regAdd, uint8_t data)

  Clear bits in a control register.
- void encDisablePowersave (void)

  Leave the power safe mode.
- void encEnablePowersave (void)

  Go to power safe mode.
- void encGetMacAdd (eth_addr_t ∗mac)

  Read the current MAC address.
- void encInit (void)

  Initialise the Ethernet controller ENC28J60.
- void encReset (void)

  Reset the Ethernet controller ENC28J60.
- void encSetBufferLimits (void)

  Set the sizes of ENC28J60's receive and transmit buffers.
- uint8_t encSetMacAdd (eth_addr_t ∗mac)

  Set the MAC address.
- void putBufferMemory (uint8_t data)

  Write one byte to ENC28J60's memory buffer.
- void readBufferMemory (uint8_t ∗dest, uint16_t n)

  Read n bytes from ENC28J60's memory buffer.
- uint8_t readControlRegister (uint8_t regAdd)

  Read a control register.
- uint16_t readPhysicalRegister (uint8_t regAdd)

  Read one of ENC28J60's physical registers.
- uint8_t setBank (uint8_t regAdd)

  Sets a register bank (in ECON1) if necessary for the given register.
- void setBitfield (uint8_t regAdd, uint8_t data)

  Set bits in a control register.
- void writeBufferMemory (uint8_t ∗source, uint16_t n)

  Write n bytes to ENC28J60's memory buffer.
- void writeControlRegister (uint8_t regAdd, uint8_t data)

  Write a control register.
- void writePhysicalRegister (uint8_t regAdd, uint16_t data)

  Write to one of ENC28J60's physical registers.

Variables

- uint16_t actPackInd

  Index of the actual receive packet in ENC's dual port memory.
- uint8_t receiveStatVec [ ]

  Receive status vector.
- uint8_t transmitStatVec [ ]

  Transmission status vector.

4.20 include/we-aut_sys/ll_system.h File Reference

4.20.1 Overview

weAutSys'/weAut_01' low level system calls and services This file contains the definitions of weAutSys'/weAut_01' low level system calls and services that are normally not used by application / user software. This is system software and must not be modified for user or application programs.

This file is part of weAutSys <weinert-automation.de>

Copyright © 2011 Albrecht Weinert, Bochum

Author

Albrecht Weinert <a-weinert.de>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Defines

- #define abortBoot

  Abort command: go to bootloader (by human command)
- #define abortHMI

  Abort command: abort by human command entry.
- #define NO_SPMIN_SAMPLE 0

  Do not sample the stack pointer and update its minimal value.
- #define PLCinRUN

  Status check: PLC is in Run.
- #define PLCinSTOP

  Status check: PLC is in Stop.
- #define PLCrun

  Abort command: run (i.e. no abort command at all)
- #define PLCstop

  Abort command: stop.
- #define PRESC_TRIM_FAC

  Prescaled time factor to millisecond.
- #define SAMPLE_MIN_SP(nonc)

  Sample the stack pointer and update its minimal value.
- #define VCO_DEFAULT 7

  Trimming of timing oscillator.
- #define WDtiOut

  Abort cause: (unexpected) watchdog timeout.

Functions

- uint8_t addr3Here (void)

  Get the upper (bit 16 ...) part of the current address.
- uint32_t addrHere (void)

  Get the current address.

- uint8_t cnt12u8_8 (void) __attribute__((pure))

  Get the 12.8 (16) µs tick value (8 bit)
- uint8_t getVCOsetting (void) __attribute__((pure))

  Trimming of timing oscillator.
- void goto_P (void ∗labelAsValue)

  Go to a label (as value) and stay within the 64Kword page.
- void initPorts (void)

  low level reset type initialisation of ports
- void initProcIO (void)

  low level reset type initialisation of process I/O
- void initStatusLeds (uint8_t LEDgnStart, uint8_t LEDrdStart)

  Initialisation of status / test LEDs (if any)
- void initSystemRes (void)

  Initialise system resources after reset or restart.
- void initSystTiming (void)

  low level clock / timer / tick initialisation
- int main (void) __attribute__((OS_main))

  The system start.
- void setAbortCommand (uint8_t command) __attribute__((always_inline))

  Set the abort cause respectively command.
- void setVCOnormal (void) __attribute__((always_inline))

  Trimming of timing oscillator.
- void slowVCOdown (void) __attribute__((always_inline))

  Trimming of timing oscillator.
- void speedVCOup (void) __attribute__((always_inline))

  Trimming of timing oscillator.

Variables

- uint8_t abortCommand

  The abort command respectively cause.
- uint8_t cn12u8

  The 12,8 µs counter summand.
- uint16_t minStckP

  Minimal stack pointer value sampled.
- uint8_t msIntTick

  The ms tick counter.
- uint8_t resetCauses

  The last reset cause(s)
- char const ∗ resetCauseText

  The last main cause as text.

4.21 include/we-aut_sys/log_streams.h File Reference

4.21.1 Overview

Buffered log stream (definitions) This file contains the implementations of weAutSys' non blocking buffered stream, mainly meant for logging. Any (standard) output stream may be decorated with this buffered stream.

According to embedded or real time requirements non-blocking has the highest priority. Hence older buffered and not yet forwarded output is allowed to be overwritten. Hence this is for logging and debugging purposes only, when un-spoiled un-blocked operation is favored over non loosing output and newer logs are more valued than older ones.

With regard the main logging or debugging purpose of this buffered stream there are some extra convenience functions for combined constant flash string output and formatting. The rationale is to reduce the temptation to use the fprint sort of output functions — usually a deadly sin in embedded real time systems.

This file is part of weAutSys <`weinert-automation.de`>

Author

Albrecht Weinert <`a-weinert.de`>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

The buffered log streams

- #define BUF_STREAMS_CAP 2047

  The buffered log streams capacity.
- #define bufLogConsIsUART

  The UART is the consumer of buffered log streams output (text) data.
- #define bufLogSetConsumer(tD)

  Set the consumer of buffered log streams output (text) data.
- #define bufLogSetUART()

  Set the UART as consumer of buffered log streams output.
- #define BufLogInBufferd

  The number of characters buffered in buffered log streams for input.
- FILE bufLogStreams

  The stream "device" being a buffer for logging.
- struct thr_data_t ∗ bufLogCons

  The consumer of buffered log streams output (text) data.
- void bufLogStreamsInit (void)

  Initialise the buffered log streams.
- void bufLogStreamsOpen (void)

  Open the buffered log streams.
- uint16_t bufLogInBufferd (FILE ∗streams)

  The number of characters buffered in buffered log streams for input.
- int bufLogGetC (void)

  Get one byte from buffered stream input.
- uint16_t bufLogGetChars (char ∗dst, uint16_t n)

  Get a number of bytes from buffered stream input.
- uint8_t uartPutLogBuf (void)

  Put characters from buffered log streams to serial output.
- uint16_t bufLogStreamsOutSpace (FILE ∗streams) __attribute__((pure))

  The buffer space available for buffered log streams output.
- uint8_t bufLogCheckOutSpace (uint16_t const reqSpace, FILE ∗const streams)

  Check the space available buffered log streams output.

- void bufLogPutC (char const c)

    Put one byte to buffered stream output.
- void bufLogPut2C (char const c1, char const c2)

    Put two bytes to buffered stream output.
- int bufLogGetChar (FILE ∗const stream)

    Get one byte from buffered stream input.
- int bufLogPutChar (char c, FILE ∗const stream)

    Put one byte to buffered stream output.
- int bufLogPutSt (char ∗src, FILE const ∗const stream)

    Put a RAM string (some characters) to buffered stream output.
- int bufLogPutSt_P (char ∗src, FILE const ∗const stream)

    Put some characters from program space to buffered stream output.

Functions

- void bufLog2Dec_P (char const ∗src, uint16_t info1, uint16_t info2)

    Log a program space string + two 3 digit decimal numbers to buffered log output.
- void bufLog4HexBE (uint16_t const info)

    Log a four digit big endian hex number to buffered log output.
- void bufLog8HexBE (uint32_t const info)

    Log an eight digit big endian hex number + one space to buffered log output.
- void bufLogDec (uint16_t const info)

    Log a 4 digit decimal number to buffered log output.
- void bufLogDec3 (uint8_t const info)

    Log a three decimal digit number with leading zeroes to buffered log output.
- void bufLogDec_P (char const ∗src, uint16_t info)

    Log a string from program space + a 4 digit decimal number to buffered log output.
- void bufLogDecB (uint8_t const info)

    Log a two decimal digit number with leading zeroes to buffered log output.
- void bufLogDecHex_P (char const ∗src, uint16_t info1, uint32_t info2)

    Log a program space string + a 3 digit decimal + an 8 digit hex number to buffered log output.
- void bufLogHex (uint8_t const info)

    Log a two digit hex number to buffered log output.
- void bufLogHex_P (char const ∗src, uint8_t info)

    Log a string from program space + a two digit hex number to buffered log output.
- void bufLogHexHex_P (char const ∗src, uint8_t info1, uint32_t info2)

    Log a program space string + a 2 digit and an 8 digit hex number to buffered log output.
- void bufLogLF (uint8_t n)

    Log space and linefeed to buffered log output.
- void bufLogMark (char ∗info, uint8_t const len)

    Output a (debug) marker text.
- void bufLogTThex_P (char const ∗tx1, char const ∗tx2, uint8_t info)

    Log two flash strings + a two digit hex number to buffered log output.
- void bufLogTxt (char ∗src, uint8_t n)

    Log some characters from RAM to buffered log output.
- void bufLogTxt_P (char const ∗src)

    Log some characters from program space to buffered log output.

4.22 include/we-aut_sys/modbus.h File Reference

4.22.1 Overview

weAutSys' system calls, services and types for the Modbus server This file contains the definitions for weAutSys' system calls, services and types for the build in Modbus server. It implies an uIP used as TCP/IP stack.

This is system software and must not be modified for user or application programs.

This file is part of weAutSys <weinert-automation.de>

Copyright © 2012 Albrecht Weinert, Bochum

Author

Albrecht Weinert <a-weinert.de>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Defines

- #define COIL_SINGLE

  Modbus data model: Single coil bit addressed 'C'.
- #define COILS_BYMAPD

  Modbus data model: Coils byte mapped 'c'.
- #define DISC_INP_BYMAPD

  Modbus data model: Discrete inputs byte mapped 'i'.
- #define HLD_MASK

  Modbus data model: Single Holding register for mask 'H'.
- #define HLD_REGISTERS

  Modbus data model: Holding registers 'h'.
- #define INP_REGISTERS

  Modbus data model: Input registers 'r'.
- #define MODB_FCIND 7

  Modbus: index of function code in the TCP/IP telegram.
- #define MODB_MBAB_LEN 7

  Modbus: length of the TCP/IP telegram start (MBAB header)

Defines for Modbus function codes and the like

This is a subset of the function codes defined for the Modbus protocol.

The others not put here are not implemented by this server.

- #define WRITE_COIL 0x05

  Modbus function code: write one bit output.
- #define READ_COILS 0x01

  Modbus function code: read back bitwise output.
- #define WRITE_COILS 0x0F

  Modbus function code: write bitwise output.
- #define READ_DISCRETE_INPUTS 0x02

  Modbus function code: read bitwise inputs.
- #define READ_INPUT_REGISTERS 0x04

Modbus function code: read word (16 bit) inputs.
- #define READ_HOLDING_REGISTERS 0x03

  Modbus function code: read back word (16 bit) outputs.
- #define WRITE_HOLDING_REGISTERS 0x10

  Modbus function code: write word (16 bit) outputs.
- #define WRITE_HOLDING_REGISTER 0x06

  Modbus function code: write (one) word (16 bit) output.
- #define MASK_WRITE_REGISTER 0x16

  Modbus function code: and and or (one) word (16 bit) output.
- #define WRITE_READ_REGISTERS 0x17

  Modbus function code: write than read word (16 bit) outputs.
- #define MODB_EXC_FUNC 1

  Modbus exception: unimplemented function code.
- #define MODB_EXC_ADDR 2

  Modbus exception: illegal address.
- #define MODB_EXC_DATA 3

  Modbus exception: invalid data (or length)
- #define MODB_EXC_OPER 4

  Modbus exception: the (partly) performed operation failed.

Functions

- ptfnct_t appModFun (struct modThr_data_t ∗m)

  Handle Modbus server events.
- ptfnct_t modbusAppcall (void)

  Handle Modbus server events.
- void modbusInit (void)

  Initialise the Modbus (server)
- void registerAppModFun (p2ptFunM appModFun)

  Register the application Modbus handler function.

4.23 include/we-aut_sys/network.h File Reference

4.23.1 Overview

weAutSys' (low level) system calls, services and types for LAN / Ethernet communication This file contains the definitions for weAutSys' (low level) system calls, services and types for LAN / Ethernet communication. They imply an ENC28J60 used for Ethernet driver anduIP used as TCP/IP stack. To change that this file and many other sources would have to be touched (and not just configuration macros).

This is system software and must not be modified for user or application programs.

Please find detailed description of the variables, functions etc. in the modules Ethernet communications and LAN communications.

This file is part of weAutSys <`weinert-automation.de`>

Author

Albrecht Weinert <`a-weinert.de`>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Data Structures

- struct ipConf_t

  The IP configuration.

Defines

- #define BUF2EH

  The uip buffer cast to Ethernet header.
- #define DHCP_MSK

  Mask for DHCP (use / set by) bit.
- #define DNS1_MSK

  Mask for DNS server 1 set bit.
- #define DNS2_MSK

  Mask for DNS server 2 set bit.
- #define DNS_CLIENT_MSK

  Mask for be DNS client bit.
- #define DNS_MSK

  Mask for DNS servers (set / use if set) bits.
- #define ECHO_PORT 7

  The well known Echo port.
- #define ENC_HWP

  Mask for the ENC HW problem bit in networkNotReady.
- #define ENC_MSK

  Mask for the ENC bits in networkNotReady.
- #define ENC_NRD

  Mask for the ENC not ready bit in networkNotReady.
- #define ENC_PWS

  Mask for the ENC powersave bit in networkNotReady.
- #define IP_ADD(addr0, addr1, addr2, addr3)

  Initializer of an IP (V4) address from four byte values.
- #define IPBUF

  The uip buffer cast to Ethernet IP header.
- #define LNC_IDN

  Mask for the link is down (at last status request) in networkNotReady.
- #define LNC_WDN

  Mask for the link was down (before last status request) in networkNotReady.
- #define MODBUS_PORT 502

  The well known Modbus port.
- #define NTP1_MSK

  Mask for NTP server 1 set bit.
- #define NTP2_MSK

  Mask for NTP server 2 set bit.
- #define NTP_CLIENT_MSK

  Mask for be NTP client bit.
- #define NTP_MSK

  Mask for NTP server bits.
- #define NTP_SERVERT_MSK

  Mask for be NTP server bit.
- #define STC_MSK

  Mask for the stack bits in networkNotReady.

- #define STC_NST

    Mask for the stack' IP not set bit in networkNotReady.
- #define TELNET_PORT 23

    The well known Telnet port.

Typedefs

- typedef struct uip_eth_addr eth_addr_t

    Representation of a 48-bit Ethernet address / MAC address.

Functions

- uint16_t deviceDriverPoll (void)

    Hand over received packages.
- void deviceDriverSend (void)

    Send the actual uIP buffer.
- char ∗ formIpAdd (char ∗s, uip_ipaddr_t ipAddr)

    Format an ipV4 Ethernet address.
- char ∗ formIpConf (char ∗s, ipConf_t ∗ipConf)

    Format an IP configuration (w/o DHCP)
- char ∗ formIpConfDHCP (char ∗s, ipConf_t ∗ipConf)

    Format an IP configuration, the DHCP part.
- char ∗ formMacAdd (char ∗s, eth_addr_t ∗mac)

    Format a MAC address.
- struct uip_eth_addr ∗ getMACforIPaddr (uip_ipaddr_t ipAddr)

    Get the MAC address for an IP address.
- void lanComStdInit (void)

    Initialise the LAN communication (standard way)
- uint8_t linkState (void)

    Check the link state.
- void networkInit (void)

    Initialise the network stack.
- void networkPolling (void)

    Receive and then perhaps send the actual uIP buffer.
- ptfnct_t outEncLanInfoThreadF (hierThr_data_t ∗threadD, FILE ∗toStream)

    Output Ethernet and driver (ENC28J60) state info, the thread function.
- uint8_t parseIpAdd (uip_ipaddr_t ipAddr, char s[ ], uint8_t si)

    Parse an ipV4 Ethernet address.
- uint8_t parseMACadd (eth_addr_t ∗macAddr, char s[ ], uint8_t si)

    Parse a MAC address.
- void setMACadd (eth_addr_t ∗mac)

    Set the IP stack's MAC address.
- void setMACaddP (const eth_addr_t ∗mac)

    Set the stack's MAC address.
- void udpAppcall (void)

    The uIP udp event function for the application software.
- void uipAppcall (void)

    The uIP event function for the application software.
- void uipGetAddresses (ipConf_t ∗ipConf)

    Get the addresses from the uIP stack.
- void uipSetAddresses (ipConf_t ∗ipConf)

    Set the addresses in the uIP stack.

Variables

- uint8_t arpTickPeriod

  The period of the ARP timeout tick in seconds.
- uint8_t arpTimeOut

  The ARP timeout counter in seconds.
- ipConf_t curIpConf

  The actual IP configuration.
- uint8_t networkNotReady

  Network not ready.

4.24 include/we-aut_sys/ntp.h File Reference

4.24.1 Overview

weAutSys' system calls and types for network time services This file contains the definitions for weAutSys' system calls, services and types for using network time. They imply an uIP used as TCP/IP stack and weAutSys' timing / time keeping modules.

Please see also the modules Ethernet communications and Time keeping (date and time).

This is system software and must not be modified for user or application programs.

This file is part of weAutSys <`weinert-automation.de`>

Copyright © 2012 Albrecht Weinert, Bochum

Author

Albrecht Weinert <`a-weinert.de`>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Data Structures

- struct ntpMess_t

  NTP message type.
- struct ntpState_t

  Structure for NTP (client) application state.
- struct ntpTimestamp_t

  NTP time stamp type.

Defines

- #define FLAGS_LI_MASK

  leap indicator bits in ntpMess_t::flags
- #define FLAGS_MODE_MASK

  mode bits in ntpMess_t::flags
- #define FLAGS_VERSION_MASK

  version bits in ntpMess_t::flags

- #define LI_59SEC

    leap indicator 2 : minute = 59s
- #define LI_61SEC

    leap indicator 1 : minute = 61s
- #define LI_ALARM

    leap indicator 3 : alarm (clock not synchronized)
- #define LI_NOWARN

    leap indicator 0 : no warning
- #define NTP_ADJUSTAT_DIRMSK

    last adjustment direction mask
- #define NTP_ADJUSTAT_MINUS

    last adjustment negative
- #define NTP_ADJUSTAT_PLUS

    last adjustment positive
- #define NTP_ADJUSTAT_PRV_MINUS

    previous (not last) adjustment negative
- #define NTP_ADJUSTAT_PRV_MSK

    previous (not last) adjustment mask
- #define NTP_ADJUSTAT_PRV_PLUS

    lprevious (not last) adjustment positive
- #define NTP_ADJUSTAT_RNG_1S

    last adjustment range +1 second
- #define NTP_ADJUSTAT_RNG_HM

    last adjustment range milliseconds
- #define NTP_ADJUSTAT_RNG_HS

    last adjustment range seconds
- #define NTP_ADJUSTAT_RNG_LM

    last adjustment range low milliseconds
- #define NTP_ADJUSTAT_RNGMSK

    last adjustment range mask
- #define NTP_BROADCAST

    NTP mode broadcast (5)
- #define NTP_CLIENT

    NTP mode client (3)
- #define NTP_FRACT_DROP

    Fraction byte significance.
- #define NTP_PORT

    The well known NTP port.
- #define NTP_SERVER

    NTP mode server (4)
- #define NTP_STATE_CNETAB 0xE0

    connection established
- #define NTP_STATE_CNPROB 0xF0

    connection problematic
- #define NTP_STATE_CONN1 1

    Server known, port connected to server 1.
- #define NTP_STATE_CONN2 2

    Server known, port connected to server 2.
- #define NTP_STATE_CONTSY 0xC0

    continuous synchronisation
- #define NTP_STATE_OPMSK

Operation (bits) mask.

- #define NTP_STATE_RESET 0

  No known NTP server, no NTP client operation.
- #define NTP_STATE_SEPROB 0xB0

  server answer problematic (bogus, alarm)
- #define NTP_STATE_SRVMSK

  Server (bits) mask.
- #define NTP_STATE_TRYREQ 0x10

  try (first) request
- #define NTP_STATE_W4REPL 8

  flag: waiting for server's reply
- #define NTP_VERSION

  current version (4) within FLAGS_VERSION_MASK
- #define ntpAsksPrio()

  NTP asks for priority.

Functions

- void adjustNTPtime308UTC (struct ntpTimestamp_t ∗ntpTimeDif)

  Adjust the actual (system) time by a NTP time stamp difference.
- ptfnct_t ntpAppcall (void)

  Handle NTP (server) events.
- void ntpInit (uint8_t pref2)

  Initialise the NTP (client)
- void ntpReset (void)

  Reset the NTP (client) to initial state.
- void ntpTimestampDif (struct ntpTimestamp_t ∗result, struct ntpTimestamp_t ∗a, struct ntpTimestamp_t ∗b)

  Difference of two NTP time stamps.
- void ntpTimestampHalf (struct ntpTimestamp_t ∗result) __attribute__((pure))

  Half a NTP time stamp value.
- void ntpTimestampSum (struct ntpTimestamp_t ∗result, struct ntpTimestamp_t ∗a, struct ntpTimestamp_t ∗b)

  Sum of two NTP time stamps.
- void ntpTimestampToMillies (uint16_t ∗ms, struct ntpTimestamp_t ∗ntpStamp)

  Milliseconds from a NTP time stamp's fraction.
- void setNTPstampAct (struct ntpTimestamp_t ∗ntpTimestamp)

  Set a NTP time stamp to actual time.

Variables

- struct ntpState_t ntpState

  the NTP state

4.25 include/we-aut_sys/persist.h File Reference

4.25.1 Overview

weAutSys (weAut_01) utility / library functions for persistent storage This file contains the definitions for weAutSys (weAut_01) utility / library functions for persistent storage. This is system software and must not be modified for user or application programs.

This file is part of weAutSys <weinert-automation.de>

Author

Albrecht Weinert <a-weinert.de>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Data Structures

- struct conf_data_t

  The device's basic configuration data type.
- struct eeOp_data_t

  The data structure for an EEPROM bulk operation thread.

Defines

- #define DEFAULT_EEPROM_CONF_ADD 64

  Default address of EEPROM configuration data.
- #define EE_CONF_ADD(elem)

  The EEPROM address of a configuration element.
- #define EEPROM_POINTER2_EE_CONF (EEP_SIZE - 64)

  EEPROM address of (address of) EEPROM configuration data.

Functions

- ptfnct_t eeOperationThread (eeOp_data_t ∗eeOpData)

  The EEPROM (bulk) write system thread; the thread function.
- uint8_t eeReadByte (uint8_t ∗readValue, uint16_t eeAddr)

  Read one EEPROM cell to RAM.
- uint8_t eeReadBytes (uint16_t eeAddr, uint8_t ∗dest, uint8_t bufferLength)

  Read EEPROM cells.
- uint8_t eeWriteByte (uint16_t eeAddr, uint8_t newValue)

  Write one EEPROM cell.
- uint8_t initEEthreadWrite (eeOp_data_t ∗eeOpData, uint16_t eeAddr, uint8_t ∗writeBuffer, uint8_t bufferLength)

  Initialise an EEPROM operation thread.
- void persistInit (void)

  Initialise persistence / EEPROM handling.

Variables

- conf_data_t defaultConfData

  The device's basic default configuration data.
- conf_data_t defaultTypeConfData

  The device type's basic default configuration data in flash memory.
- uint16_t eeConfigAdd

  Address of EEPROM configuration data.
- eeOp_data_t eeOpData

  The data for an EEPROM bulk operation thread.

4.26 include/we-aut_sys/proc_io.h File Reference

4.26.1 Overview

weAutSys'/weAut_01's (low level) system calls and services for process I/O and HMI This file contains the definitions for weAutSys / weAut_01 (low level) system calls and services for process input and output (I/O) and I/O for the human machine interface (HMI). This is system software and must not be modified for user or application programs.

This file is part of weAutSys <weinert-automation.de>

Copyright © 2011 Albrecht Weinert, Bochum

Author

Albrecht Weinert <a-weinert.de>

Revision:

4

Date:

2014-09-20 10:22:58 +0200 (Sa, 20 Sep 2014)

Defines

- #define enterKeyPressed(X)

  Enter key is (stable) pressed.
- #define enterKeyRel(X)

  Enter key is released.
- #define enterKeyReleased(X)

  Enter key is (stable) released.

Functions

- uint8_t actDI (void) __attribute__((always_inline))

  Actual digital (process) input DI.
- uint8_t actDiLEDs (void) __attribute__((always_inline))

  DI display LEDs: actual value.
- uint8_t actDOdriver (void) __attribute__((always_inline))

  DO digital (process) output: actual value.
- uint8_t dctDI (void) __attribute__((always_inline))

  Digital (process) input DI (direct)

- uint8_t doDriverEnabled (void) __attribute__((always_inline))

  Digital (process) output DO driver enabled.
- uint8_t doDriverOK (void) __attribute__((always_inline))

  Digital (process) output DO driver OK.
- void enableDOdriver (uint8_t state) __attribute__((always_inline))

  Enable the digital (process) output DO driver.
- uint8_t filDI (void) __attribute__((always_inline))

  The final or filtered digital (process) input DI.
- uint8_t lbpDI (void) __attribute__((always_inline))

  Last before previous digital (process) input DI.
- void offDiLEDs (void) __attribute__((always_inline))

  Turn DI display LEDs off.
- void onDiLEDs (void) __attribute__((always_inline))

  Turn DI display LEDs on.
- void procDIcyc (void)

  Digital (process) input DI (system implementation)
- uint8_t prvDI (void) __attribute__((always_inline))

  Previous digital (process) input DI.
- uint8_t prvDiLEDs (void) __attribute__((always_inline))

  DI display LEDs: previous value.
- uint8_t prvDOdriver (void) __attribute__((always_inline))

  DO digital (process) output: previous value.
- void setAIchannels (uint8_t mask)

  Set the usage of channel(s) as AI instead of DI.
- void toDiLEDs (uint8_t value)

  Output to DI display LEDs.
- void toDOdriver (uint8_t value)

  Output to digital (process) output DO.

Variables

- uint8_t aiChannels

  Use channel(s) as AI instead of DI.
- uint8_t aiConvd

  Analogue input available.
- uint8_t aiResult [8]

  Analogue input results.
- uint8_t lowLV

  Load voltage low.
- uint8_t pbFil

  Filtered Port B input.
- uint8_t upDIthresh4hyst

  Shift DI thresholds up mask conditionally / larger hysteresis.
- uint8_t upDIthreshForce

  Shift DI thresholds up mask permanently.

4.27 include/we-aut_sys/smc.h File Reference

4.27.1 Overview

weAutSys' (low level) system calls, services and types for communication with a small memory card This file contains the definitions for weAutSys' (low level) system calls, services and types to communicate via SPI with a small memory card.

This is system software and must not be modified for user or application programs.

This file is part of weAutSys <weinert-automation.de>

Copyright © 2012 Albrecht Weinert, Bochum

Author

Albrecht Weinert <a-weinert.de>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Data Structures

- struct smcThr_data_t

  The organisational data for a small memory card (SMC) handling thread.

Defines

- #define APP_CMD 55

  SMC command: next command is application specific.
- #define CT_HC 0x08

  Card type flag: high capacity.
- #define CT_INS_POWD 0x30

  Card type mask: power & inserted.
- #define CT_INSERTED 0x20

  Card type flag: inserted.
- #define CT_OPCN_MASK 0xF0

  Card type mask: operation bits.
- #define CT_POWERD_UP 0x10

  Card type flag: was powered up.
- #define CT_TYPE_MASK 0x0F

  Card type mask: type bits.
- #define CT_V_1 0x02

  Card type: SD version 2.
- #define CT_V_2 0x04

  Card type: SD version 2.
- #define CT_V_3 0x01

  Card type: MMC version 3.
- #define FL_CAT 0x40

  Action/state flag: card type.
- #define FL_IMP 0x76

Action mask: implemented actions (except Bit 7)
- #define FL_INI 0xF0

  Action mask: initialisation actions.
- #define FL_R_W 0x06

  Action mask: sector access (read, write)
- #define FL_RDS 0x02

  Action/state flag: sector read.
- #define FL_WRS 0x04

  Action/state flag: sector write.
- #define GO_IDLE_STATE 0

  SMC command: soft reset.
- #define LOCK_UNLOCK 42

  SMC command: lock / unlock the card.
- #define NCR_EXTR_WAIT 8

  Maximum extra waits for command response.
- #define othersAskPrio()

  Other devices asks for priority.
- #define READ_OCR 58

  SMC command: read the card's OCR register (R3)
- #define READ_SINGLE_BLOCK 17

  SMC command: read one block.
- #define SEND_CID

  SMC command: get CID.
- #define SEND_CID

  SMC command: get CID.
- #define SEND_CSD 9

  SMC command: send card specific data.
- #define SEND_IF_COND 8

  SMC command: send interface condition.
- #define SEND_OP_COND 1

  SMC command: init. process.
- #define SEND_OP_COND_APP

  SMC application specific command: initialisation process.
- #define SEND_STATUS 13

  SMC command: get status (R2)
- #define SET_BLOCKLEN 16

  SMC command: set block length for LOCK_UNLOCK.
- #define SMC_DAT_TOK 0xFE

  data token
- #define SMC_LAST_R1

  Last command response - first or single (R1) byte.
- #define SMC_SEC_BUF_SZ 516

  Sector buffer size.
- #define smcInsPow()

  Inserted card is powered up.
- #define smcReceive()

  Receive a single byte from the small memory card.
- #define smcReceiveN(receiveB, skip, n)

  Receive n bytes from the SMC to a buffer with optional skip.
- #define smcTypeD()

  The SMC's type is determined and OCR is known.

- #define smcXmit(datByte)

  Transmit a single byte to the small memory card.
- #define smcXmit2(sendB1, sendB2)

  Send two bytes and receive one byte to/from the small memory card.
- #define WRITE_BLOCK 24

  SMC command: write one block.

Functions

- uint8_t checkBusy (uint8_t tries)

  Check if card is busy.
- void clk80 (void)

  Have 80 dummy SPI clocks.
- uint8_t crc7stp (uint8_t crcIn, uint8_t datByte) __attribute__((always_inline))

  Calculate CRC7 (one step)
- void deSelectSMC (void) __attribute__((always_inline))

  De-select the small memory card.
- uint8_t doSectorRead (uint32_t sector)

  Order sector read (as background task)
- uint8_t doSectorSync (void)

  Order sector synchronisation (as background task)
- uint8_t getSMCtype (void)

  Determine the card type.
- void initSMCthreadState (uint8_t actionFlag)

  Initialise the small memory card (smc) handling thread.
- uint8_t readDataBlock (uint32_t sector, uint8_t ∗buff)

  Read single data block (512 byte)
- uint8_t sendAppCmd (uint8_t appCmdNum, uint32_t cmdArg)

  Send an application specific command to the small memory card.
- uint8_t sendCmd (uint8_t cmdNum, uint32_t cmdArg)

  Send a command to the small memory card.
- uint8_t sendCmd0arg (uint8_t cmdNum)

  Send a command with 0 argument to the small memory card.
- void setSectorModified (uint32_t sector)

  Mark sector buffer data as modified (start)
- uint32_t smcGetSectCount (void)

  Get the sector count.
- uint8_t smcInsertSwitch (void)

  Hardware card detect.
- uint8_t smcReadCID (uint8_t ∗buff)

  Read the SMC's CID.
- uint8_t smcReadCSD (void)

  Read the SMC's CSD.
- uint8_t smcReadOCR (void)

  Read the SMC's ORC.
- void smcSetFrq (uint8_t frqUse) __attribute__((always_inline))

  Set the SPI clock frequency for the small memory card.
- uint8_t smcSetIdle (uint8_t mode)

  Put card to idle state.
- ptfnct_t smcThreadF (void)

  The small memory card (smc) handling thread.
- uint8_t writeDataBlock (uint32_t sector, const uint8_t ∗buff)

  Write single data block (512 byte)

Variables

- struct smcThr_data_t smcState

  The (one) small memory card (SMC) state.

4.27.2 Define Documentation

4.27.2.1 #define SMC_LAST_R1

Last command response - first or single (R1) byte.

See also

lastCmdResp

4.28 include/we-aut_sys/smc2fs.h File Reference

4.28.1 Overview

weAutSys' file system adaption to a small memory card This file contains the definitions for weAutSys' adaption of ChaN's fatFS to the (low level) services to communicate with a small memory card (SMC) via SPI.

This is system software and must not be modified for user or application programs.

This file is part of weAutSys <`weinert-automation.de`>

Copyright © 2012 Albrecht Weinert, Bochum

Author

Albrecht Weinert <`a-weinert.de`>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Data Structures

- struct FS_WORK

  Work space for file system operations (structure FS_WORK)

Defines

- #define fMountSMC()

  Mount / initialise the SMC file system as drive 0.
- #define lockFsWorkFor(ls)

  Set the lock on fsWork.
- #define SMC_FS_CLIUSE 0xCB

  SMC file system (structure) locked for (application) CLI.
- #define SMC_FS_SYSTUSE 0xFB

  SMC file system (structure) locked for runtime / system use.
- #define unlockFsWorkFrom(ls)

  Unset the lock on fsWork.

Functions

- char ∗ formFATdate (char ∗s, uint16_t fatDate)

  Format a FAT date.
- char ∗ formFATtime (char ∗s, uint16_t fatTime)

  Format a FAT time.
- uint8_t stdPutFilInf (FILINFO ∗filInf, const uint8_t inf)

  Write file info data to standard output.

Variables

- FATFS fileSystSMC

  File system object (the one for SMC)
- FS_WORK fsWork

  Work space for file system operations.

4.29 include/we-aut_sys/spi.h File Reference

4.29.1 Overview

weAutSys'/weAut_01' system calls and services for the SPI interfaces This file contains the definitions for weAutSys / weAut_01 (low level) system calls and services for the serial peripheral interfaces (SPI). This is system software and must not be modified for user or application programs.

This file is part of weAutSys <`weinert-automation.de`>

Copyright © 2011 Albrecht Weinert, Bochum

Author

Albrecht Weinert <`a-weinert.de`>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Defines

- #define memCardIsSelected()

  The memory card (inserted) is chip-selected.
- #define nicIsSelected()

  The Ethernet driver is chip-selected.
- #define SPI100kHz

  control value to set SPI 2 clock to 100 kHz
- #define SPI10MHz 0

  Divisor register for UART as SPI (UBRR) setting for 10 MHz.
- #define SPI1MHz

  control value to set SPI 2 clock to 1 MHz
- #define SPI200kHz

  control value to set SPI 2 clock to 200 kHz

- #define SPI2COND_ERR_RET(cond, err, ret)

    Check a SPI2 condition with timeout.
- #define SPI2MHz

    UBRR value to set SPI clock to 2 MHz.
- #define SPI400kHz

    control value to set SPI 2 clock to 400 kHz
- #define SPI5MHz 1

    Divisor register for UART as SPI (UBRR)) setting for 5 MHz.
- #define UCPHA1

    UCSR1C register (bit number)

Functions

- uint8_t deSelectSPI1 (void) __attribute__((always_inline))

    De-select all SPI1 devices.
- void deSelectSPI1_impl (void) __attribute__((always_inline))

    De-select all SPI1 devices (raw)
- void deSelectSPI2 (void) __attribute__((always_inline))

    De-select all SPI2 devices.
- void selectDIdisplayLEDs (void) __attribute__((always_inline))

    Chip-select a SPI 1 device: the digital input display LEDS (HMI)
- void selectDOdrivDIleds (void) __attribute__((always_inline))

    Chip-select two SPI 1 devices: the DO driver and the DI LEDs (HMI)
- void selectDOdriver (void) __attribute__((always_inline))

    Chip-select a SPI 1 device: the digital output driver (DO)
- void selectEtherChip (void) __attribute__((always_inline))

    Chip-select a SPI 2 device: the Ethernet driver chip.
- void selectMemCard (void) __attribute__((always_inline))

    Chip-select a SPI 2 device: the memory card (holder)
- uint8_t spi2Receive (void) __attribute__((always_inline))

    Receive one byte from SPI 2.
- void spi2ReceiveNS (uint8_t ∗receiveB, uint16_t n, uint8_t skip)

    Receive n bytes to a buffer via SPI 2 with optional skip (afterwards)
- void spi2ReceiveSN (uint8_t ∗receiveB, uint16_t skip, uint16_t n)

    Receive n bytes to a buffer via SPI 2 with optional skip (before)
- uint8_t spi2Tranceive (uint8_t sendB) __attribute__((always_inline))

    Send and receive one byte via SPI 2.
- uint8_t spi2Tranceive2 (uint8_t sendB1, uint8_t sendB2)

    Send two bytes and receive one byte via SPI 2.
- uint8_t spi2TranceiveN (uint8_t sendB, uint8_t n)

    Send a byte n times and receive one byte via SPI 2.
- void spi2Transmit (uint8_t sendB) __attribute__((always_inline))

    Send one byte over SPI 2.

Variables

- uint8_t spi2Errors

    Accumulated errors at SPI 2.
- uint8_t spi2EtherChipBaud

    The SPI (2) clock frequency used for the Ethernet driver chip.
- uint8_t spi2MemCardBaud

    The SPI (2) clock frequency used for the memory card (holder)

4.30 include/we-aut_sys/streams.h File Reference

4.30.1 Overview

Streams (definitions) This file contains the definitions for weAutSys' I/O via (standard i.e. stdio like) streams. It is system software and must not be modified for user or application programs.

This file is part of weAutSys <weinert-automation.de>

Author

Albrecht Weinert <a-weinert.de>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Data Structures

- struct streamClassDescript_t

  Extra stream (type) specific data.

Defines

- #define AVAILABLE_UNKNOWN 0xFFFF

  Available input unknown.
- #define LAN_IS_STDOUT

  stdout is the LAN stream output (Ethernet, uIP)
- #define SER_IS_STDOUT

  stdout is the serial output (UART)
- #define USE_AVR_GCC_FILE 1

  use AVR FILE type internals

Common stream handling definitions

- #define PT_YIELD_OUT_SPACE(pt, requSpace, streams)

  Set standard I/O and exit or yield for requested output space.
- #define PT_WAIT_OUT_SPACE(pt, requSpace, streams)

  Set standard I/O and exit or wait for requested output space.
- typedef uint16_t(∗ str2InBufferd)(const FILE ∗stream)

  Function pointer / type: number of characters buffered for input.
- typedef uint16_t(∗ str2OutSpace)(const FILE ∗stream)

  Function pointer / type: buffer space available for output.
- typedef uint8_t(∗ str2CheckOutSpace)(uint16_t reqSpace, const FILE ∗stream)

  Function pointer / type: checking the space available for output.
- typedef int(∗ str2PutS)(char ∗src, const FILE ∗stream)

  Function pointer / type: writing a RAM string to output.
- typedef int(∗ str2PutS_P)(prog_char ∗src, const FILE ∗stream)

Function pointer / type: writing a flash memory string to output.

- typedef struct streamClassDescript_t streamClassDescript_s

  Extra stream (type) specific data (structure)
- uint16_t inBufferd (const FILE ∗stream)

  Number of characters buffered for input.
- uint16_t outSpace (const FILE ∗stream)

  Buffer space available for output.
- uint8_t checkOutSpace (uint16_t reqSpace, const FILE ∗stream)

  Checking the space available for output.
- int putSt (char ∗src, const FILE ∗stream)

  Write a RAM string to output.
- int putSt_P (char const ∗src, const FILE ∗stream)

  Write a flash memory string to output.

The serial streams

- FILE serStreams

  The stream "device" connected to the serial IO / UART(0)
- void serStreamsOpen (void)

  Open the serial (UART) streams.
- uint16_t serStreamsOutSpace (FILE ∗streams)

  The buffer space available for serial output.
- uint8_t serCheckOutSpace (uint16_t reqSpace, FILE ∗streams)

  Check the space available for serial output.
- int serPutChar (char c, FILE ∗stream)

  Put one byte to serial output.

Recommended standard I/O functions

- #define stdPutS(src)

  Write a RAM string to `stdout`*.*
- #define stdPutS_P(src)

  Write a flash memory string to `stdout`*.*
- void stdPutC (char c)

  Write a character to `stdout`*.*
- void logStackS (uint8_t maxBytes)

  Log the stack frame to stdout.

The Ethernet / LAN streams

- FILE lanStreams

  The stream "device" connected to the Ethernet (uIP)
- void lanStreamsInit (void)

  Initialise the Ethernet / LAN streams.
- void lanStreamsEcho (void)

  Set the Ethernet / LAN streams to echo input.
- void lanStreamsOpen (void)

  Open the Ethernet / LAN streams.
- void lanStreamSend (void)

Sends the LAN streams data (over the Ethernet)
- uint16_t lanStreamsOutSpace (FILE ∗streams)

  The buffer space available for Ethernet / LAN streams output.
- uint8_t lanCheckOutSpace (uint16_t reqSpace, FILE ∗streams)

  Check the space available Ethernet / LAN streams output.
- uint16_t lanInBufferd (FILE ∗streams)

  The number of characters buffered from LAN (stream) input.
- int lanGetC (void)

  Get one byte from LAN stream input.
- void lanPutC (char c)

  Put one byte to LAN stream output.
- int lanGetChar (FILE ∗stream)

  Get one byte from LAN stream input.
- int lanPutChar (char c, FILE ∗stream)

  Put one byte to LAN stream output.
- int lanPutChars (char ∗src, uint16_t n)

  Put some characters to LAN stream output.
- int lanPutSt (char ∗src, const FILE ∗stream)

  Put a RAM string (some characters) to LAN stream output.
- int lanPutSt_P (char ∗src, const FILE ∗stream)

  Put some characters from program space to LAN stream output.

The nul device

- FILE nulStreams

  The stream "device" connected to the nul device.
- void nulStreamsOpen (void)

  Open the nul streams.
- uint16_t nulStreamsOutSpace (FILE ∗streams)

  The buffer space available for nul output.
- uint8_t nulCheckOutSpace (uint16_t reqSpace, FILE ∗streams)

  Check the space available for nul streams output.
- uint16_t nulInBufferd (FILE ∗streams)

  The number of characters buffered from nul input.
- int nulGetC (void)

  Get one byte from nul stream input.
- void nulPutC (char c)

  Put one byte to nul stream output.
- int nulPutChar (char c, FILE ∗stream)

  Put one byte to nul stream output.
- int nulGetChar (FILE ∗stream)

  Get one byte from nul stream input.
- int nulPutSt (char ∗const src, const FILE ∗stream)

  Put a string (some characters) to nul stream output.

Functions

- void initStdStreams (FILE ∗initStream)

  Initialise the standard streams.
- void streamsClose (void)

  Close other standard streams.
- uint8_t switchStdStreams (FILE ∗toStream)

  Switch the standard streams to another stream.

4.31 include/we-aut_sys/syst_threads.h File Reference

4.31.1 Overview

weAutSys' system threads This file contains the definitions for weAutSys' system threads. Please find detailed description of the variables, functions etc. in the module system threads.

This is system software and must not be modified for user or application programs.

This file is part of weAutSys <weinert-automation.de>

Copyright © 2014 Albrecht Weinert, Bochum

Author

Albrecht Weinert <a-weinert.de>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Defines

- #define ABSsecADJ 0x80

  absolute date time seconds were adjusted
- #define ABSsecSET 0x40

  absolute date time seconds were set
- #define exitNotFlaggedThread(thread)

  Exit a not flagged user / application thread.
- #define resetAppThread(thread)

  Reset respectively initialise an user / application thread.
- #define runOutSysInfoThread(parent, pt, toStream)

  Start (spawn) the outSysInfoThreadF thread as child.
- #define scheduleAppThread(thread)

  Unconditionally schedule an user / application thread.
- #define scheduleFlgdAppThread(thread)

  Schedule a flagged user / application thread.
- #define scheduleRegdAppThread(thread)

  Schedule a registered user / application thread.
- #define scheduleYldAppThread(thread)

  Schedule a flagged or yielding user / application thread.
- #define SINCsINCR 0x01

  run since seconds were incremented

Functions

- ptfnct_t app100msThreadF (struct mThr_data_t ∗uthr_data)

  The user / application specific 100 ms thread.
- ptfnct_t app10msThreadF (struct mThr_data_t ∗uthr_data)

  The user / application specific 10 ms thread.
- ptfnct_t app1msThreadF (struct mThr_data_t ∗uthr_data)

The user / application specific 1 ms thread.
- ptfnct_t app1sThreadF (struct mThr_data_t ∗uthr_data)

  The user / application specific one second thread.
- ptfnct_t appBgTskThreadF (void)

  The user / application specific background task thread (the function)
- ptfnct_t appInitThreadF (struct pt ∗pt)

  The user / application specific initialisation thread.
- ptfnct_t appSerInpThreadF (struct thr_data_t ∗serInpThread)

  The user / application serial input processing thread.
- void initOutFlashTextThread (outFlashTextThr_data_t ∗outFlashTextThread, char const ∗const ∗theFlashStrings2out, uint8_t noOfFlashStrings2out)

  Initialise the output a flash string array to standard output thread.
- void initSetAppThread (struct mThr_data_t ∗thread, p2ptFun threadF)

  Register and (force) init an user / application thread.
- void initThreads (void)

  Initialise the threads.
- ptfnct_t outFlashTextThreadF (outFlashTextThr_data_t ∗outFlashTextThread, FILE ∗toStream)

  Output a flash string array to standard output thread.
- ptfnct_t outSysInfoThreadF (pt_t ∗pt, FILE ∗toStream)

  Output some system (version) info, the thread function.
- p2ptFun registerAppBgTskThread (p2ptFun threadF)

  Register a user / application background task thread.
- p2ptFunC registerAppSerInpThread (p2ptFunC threadF)

  Register a user / application serial input processing thread.
- p2ptFun registerAppThread (struct mThr_data_t ∗thread, p2ptFun threadF)

  Register an user / application thread.
- ptfnct_t sys100msThread (void)

  The 100 milliseconds system thread, the function.
- void sys1msThread (void)

  The milliseconds system function (or thread)
- ptfnct_t sys1sThread (void)

  The 1 second system thread, the function.

Variables

- struct mThr_data_t app100msThread

  The user / application 100ms thread.
- struct mThr_data_t app10msThread

  The user / application 10ms thread.
- struct mThr_data_t app1msThread

  The user / application 1ms thread.
- struct mThr_data_t app1sThread

  The user / application 100ms thread.
- struct mThr_data_t appBgTskThread

  The user / application specific background task thread.
- struct thr_data_t appSerInpThread

  The user / application serial input handling thread.
- struct pt ptSys100msThread

  The 100 milliseconds system thread, the (raw) Protothreads datastructure.
- struct pt ptSys1sThread

The 1 second system thread, the (raw) Protothreads datastructure.

- uint8_t sys100msPeriodFlag

  Flag for system 100ms thread.
- uint8_t sys1sPeriodFlag

  Flag for system 1s thread.

4.32 include/we-aut_sys/system.h File Reference

4.32.1 Overview

weAutSys definition of system calls and services to be used by application / user software This is system software and must not be modified for user or application programs.

This file is part of weAutSys <weinert-automation.de>

Copyright © 2011 Albrecht Weinert, Bochum

Author

Albrecht Weinert <a-weinert.de>

Revision:

14

Date:

2015-08-04 15:16:04 +0200 (Di, 04 Aug 2015)

Defines

- #define comAutomationNum

  command: [-stop | -start] PLC cycles (flash text)
- #define comBootNum

  command: boot -load | -reset (flash text)
- #define comDateNum

  command: show / set (local) date
- #define comDHCPNum

  command: show / restart DHCP
- #define comDirlistNum

  command: dirList [-option] [dirPath] memory card directory
- #define comDNSNum

  command: resolve a name by DNS
- #define comENCNum

  command: ENC28J60 network interface (same as NICont)
- #define comEthAddrNum

  command: show address (ARP) info (flash text)
- #define comHelpNum

  command: show help (flash text)
- #define comIPconfigNum

  command: show IP configuration info (flash text)
- #define comIPdefAdNum

  command: show / set IP address (flash text)
- #define comMacAddrNum

command: show / set MAC address (flash text)
- #define comModbusNum

  command: modbus [-stop -off -reset | ...
- #define comNICNum

  command: NICont [-off | -on | -reset | -restart]
- #define comNTPNum

  command: show / set NTP state and configuration
- #define comPLContrlNum

  command: [-stop | -start] PLC cycles (flash text)
- #define comRunInfoNum

  command: show run time info (flash text)
- #define comSMCNum

  command: SMCard [-on | -start | reset | -rest...
- #define comTelnetNum

  command: telnet [-stop]
- #define comTimeNum

  command: show / set (local) time
- #define comTypeFileNum

  command: typeFile filename display a file
- #define comUARTNum

  command: show / set UART flow control (flash text)
- #define comVersionNum

  command: show version info (flash text)
- #define comWatchDNum

  command: watchdog [abort | sharp | lenient | normal]
- #define comWDogNum

  command: wdog 60 ms | 120 ms | 250 ms
- #define comZoneNum

  command: show / set time zone and DST state
- #define optAbortNum

  options: abort a hard reset
- #define optAlternNum

  options: use or restart with alternative
- #define optAmbigousNum

  options: option given ambiguously
- #define optApplicNum

  options: application related
- #define optDebugNum

  options: debug lengthy output for trouble shooting
- #define optFastNum

  options: fast fast(er) speed
- #define optFlowCnNum

  options: ask or set flow control behaviour
- #define optHelpNum

  options: help command specific help
- #define optInfoNum

  options: inform normal / informing output
- #define optLenientNum

  options: lenient more forgiving
- #define optLoadNum

  options: perform load operation or set load mode

- #define optLogHereNum

  options: use this device as log output
- #define optNoLogsNum

  options: do not use this device as log output
- #define optNormalNum

  options: normal standard behaviour / speed
- #define optNotGivenNum

  options, commands, parameters: no option given / set
- #define optOffNum

  options: off turn off / stop
- #define optOnNum

  options: on turn on / start
- #define optOptionNum

  options: options related
- #define optPrimarNum

  options: use or restart with primary resource
- #define optQuestNum

  options: ? like -help
- #define optQuietNum

  options: quiet like -silent
- #define optReadNum

  options: perform read operation or set read mode
- #define optResetNum

  options: reset like -stop or -restart (it depends)
- #define optRestartNum

  options: restart like -start (in most cases)
- #define optSharpNum

  options: sharp more exact / less forgiving
- #define optSilentNum

  options: silent no or only urgent output
- #define optSlowNum

  options: slow slow(er) speed
- #define optStartNum

  options: start like -on (in most cases)
- #define optStopNum

  options: stop like -off (in most cases)
- #define optSystemNum

  options: system related
- #define optVerboseNum

  options: verbose lengthy output
- #define optWriteNum

  options: perform write operation or set write mode
- #define SYST_DAT "$Date: 2015-08-04 15:16:04 +0200 (Di, 04 Aug 2015) $"

  This file's last modification date (SVN)
- #define SYST_MOD

  This file's last modifier respectively SVN author.
- #define SYST_NAM "weAutSys"

  The runtime's name.
- #define SYST_REV

  The runtime's revision.

Functions

- char ∗ getSomeCharsP (char ∗dst, char const ∗src, uint8_t mxLen)

  Copy some characters from program space to a string.
- void initTestPins (uint8_t TP0start, uint8_t TP1start)

  Initialisation of the (potential) test pins.
- void setStatusLedGn (uint8_t state)

  Set the state of the green (test) LED.
- void setStatusLedRd (uint8_t state)

  Set the state of the red (test) LED.
- void setTestPin0 (uint8_t state) __attribute__((always_inline))

  Set the state of Test-Pin 0.
- void setTestPin1 (uint8_t state) __attribute__((always_inline))

  Set the state of Test-Pin 1.
- void toggleStatusLedGn (void)

  Change the state of the green (test) LED.
- void toggleStatusLedRd (void)

  Change the state of the red (test) LED.

Variables

- char const arpEmpty [ ]

  " no valid entries in ARP table \n\n" flash text building block
- char const const arpEntries [ ]

  " ARP entries " flash text building block
- char const comAutomation [ ]

  command: [-stop | -start] PLC cycles (flash text)
- char const comBoot [ ]

  command: boot -load | -reset (flash text)
- char const comDate [ ]

  command: show / set (local) date
- char const comDHCP [ ]

  command: show / restart DHCP
- char const comDirlist [ ]

  command: dirList [-option] [dirPath] memory card directory
- char const comDNS [ ]

  command: resolve a name by DNS
- char const comENC [ ]

  command: ENC28J60 network interface (same as NICont)
- char const comEthAddr [ ]

  command: show address (ARP) info (flash text)
- char const comHelp [ ]

  command: show help (flash text)
- char const comIPconfig [ ]

  command: show IP configuration info (flash text)
- char const comIPdefAd [ ]

  command: show / set IP address (flash text)
- char const comMacAddr [ ]

  command: show / set MAC address (flash text)
- char const commAmbig [ ]

  command report: ambiguous (flash text)

- char const comModbus [ ]

  command: modbus [-stop -off -reset | ...
- char const commWrong [ ]

  command report: wrong (flash text)
- char const comNIC [ ]

  command: NICont [-off | -on | -reset | -restart]
- char const comNTP [ ]

  command: show / set NTP state and configuration
- char const comPLContrl [ ]

  command: [-stop | -start] PLC cycles (flash text)
- char const comRunInfo [ ]

  command: show run time info (flash text)
- char const comSMC [ ]

  command: SMCard [-on | -start | reset | -rest...
- char const comTelnet [ ]

  command: telnet [-stop] : Telnet [close]
- char const comTime [ ]

  command: show / set (local) time
- char const comTypeFile [ ]

  command: typeFile filename display a file
- char const comUART [ ]

  command: show / set UART options
- char const comVersion [ ]

  command: show version info (flash text)
- char const comWatchD [ ]

  command: watchdog [abort | sharp | lenient | normal]
- char const comWDog [ ]

  command: wdog 60 ms | 120 ms | 250 ms
- char const comZone [ ]

  command: show / set time zone and DST state
- char const encEthSta [ ]

  " ENC/Eth st. : " flash text building block
- char const helpHeader [ ]

  Headline for system commands overview (help list)
- char const helpUserCm [ ]

  Headline for application commands overview (help list)
- char const ip4Add_is [ ]

  " IP4 address : " flash text building block
- char const ipConDefR [ ]

  " def. router : " flash text building block
- char const ipConDefS [ ]

  " def. set " flash text building block
- char const ipConDHCs [ ]

  " DHCP set " flash text building block
- char const ipConDHCu [ ]

  " use DHCP " flash text building block
- char const ipConDNSa [ ]

  " DNS address : " flash text building block
- char const ipConIpAd [ ]

  " IP4 address : " flash text building block
- char const ipConNMsk [ ]

" IP4 netmask : " flash text building block

- char const ipConNTPa [ ]

  " NTP address : " flash text building block
- char const ipDefault [ ]

  " IP4 default : " flash text building block
- char const const macAdd_is [ ]

  " MAC address : " flash text building block
- char const noUserCLI [ ]

  report: no user CLI (flash text)
- char const optAbort [ ]

  options: abort a hard reset
- char const optAltern [ ]

  options: use or restart with alternative
- char const optAmbig [ ]

  option report: ambiguous (flash text)
- char const optApplic [ ]

  options: application related
- char const optDebug [ ]

  options: debug lengthy output for trouble shooting
- char const optFast [ ]

  options: fast fast(er) speed
  ";
- char const optFlowCn [ ]

  options: ask or set flow control behaviour
- char const optHelp [ ]

  options: help command specific help
- char const optInfo [ ]

  options: inform normal / informing output
- char const optionHeader [ ]

  options: ∗ ∗ ∗ *Command options*
- char const optLenient [ ]

  options: lenient more forgiving
- char const optLoad [ ]

  options: perform load operation or set load mode
- char const optLogHere [ ]

  options: use this device as log output
- char const optNoLogs [ ]

  options: do not use this device as log output
- char const optNormal [ ]

  options: normal standard behaviour / speed
- char const optOff [ ]

  options: off turn off / stop
- char const optOn [ ]

  options: on turn on / start
- char const optOption [ ]

  options: options related
- char const optPrimar [ ]

  options: use or restart with primary resource
- char const optQuest [ ]

  options: ? like -help
- char const optQuiet [ ]

options: quiet like -silent
- char const optRead [ ]

    options: perform read operation or set read mode
- char const optReset [ ]

    options: reset like -stop or -restart (it depends)
- char const optRestart [ ]

    options: restart like -start (in most cases)
- char const optSharp [ ]

    options: sharp more exact / less forgiving
- char const optSilent [ ]

    options: silent no or only urgent output
- char const optSlow [ ]

    options: slow slow(er) speed
    ";
- char const optStart [ ]

    options: start like -on (in most cases)
- char const optStop [ ]

    options: stop like -off (in most cases)
- char const optSystem [ ]

    options: system related
- char const optVerbose [ ]

    options: verbose lengthy output
- char const optWrite [ ]

    options: perform write operation or set write mode
- char const optWrong [ ]

    option report: wrong (flash text)
- char const sepLoB [ ]

    Blanks and left opening brace.
- char const sepRcB [ ]

    Right closing brace and blanks.
- char const sysRunSectorN [ ]

    report: LF SMC sector buffered: 0x
- char const systAut [ ]

    The author of weAutSys.
- char const systBld [ ]

    The build date and time.
- char const systBye [ ]

    An farewell (abort) line with three leading feeds and the system name.
- char const systCop [ ]

    weAutSys's copyright notice.
- char const systDat [ ]

    The system's last modification date.
- char const ∗const systemCommands [ ]

    flash array of the (flash) system command definitions
- char const ∗const systemOptions [ ]

    List and definitions of command options.
- char const systMod [ ]

    The system's last modifier.
- char const systNam [ ]

    The name of the runtime system weAutSys.
- char const systRev [ ]

The system's revision.

- char const systWlc [ ]

  A greeting line with three leading feeds and the system name.
- char const wdSetLeni [ ]

  report: watchdog set long / lenient (flash text)r
- char const wdSetNormal [ ]

  report: watchdog set normal (flash text)r
- char const wdSetSharp [ ]

  report: watchdog set sharp (flash text)
- char const wdSetShort [ ]

  report: watchdog set short (flash text)

4.33 include/we-aut_sys/telnet.h File Reference

4.33.1 Overview

weAutSys' system calls, services and types for the Telnet server This file contains the definitions for weAutSys' system calls, services and types for the build in Telnet server. It implies an uIP used as TCP/IP stack.

This is system software and must not be modified for user or application programs.

This file is part of weAutSys <weinert-automation.de>

Copyright © 2012 Albrecht Weinert, Bochum

Author

Albrecht Weinert <a-weinert.de>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Defines

Defines for Telnet command, state and options

This is a subset of the command and option codes defined for the Telnet protocol.

The use of the others not put here is discouraged.

- #define IAC

  Telnet: interpret as command (IAC)
- #define DONT 254

  Telnet: option negotiation command respectively answer.
- #define DO 253

  Telnet: option negotiation command respectively answer.
- #define WONT 252

  Telnet: option negotiation command respectively answer.
- #define WILL 251

  Telnet: option negotiation command respectively answer.
- #define LINEMODE 34

  Telnet option code.
- #define SUPPRESS_GA 3

  Telnet option suppress go ahead.

- #define TEL_SB 250

  Telnet command opening brace SB.
- #define TEL_SE 240

  Telnet command closing brace SE.

Functions

- ptfnct_t telnetAppcall (void)

  Handle Telnet server events.
- void telnetInit (void)

  Initialise the Telnet (server)

4.34 include/we-aut_sys/timing.h File Reference

4.34.1 Overview

weAutSys definition of timing services This is system software and must not be modified for user or application programs.

This file is part of weAutSys <weinert-automation.de>

Copyright © 2011 Albrecht Weinert, Bochum

Author

Albrecht Weinert <a-weinert.de>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Data Structures

- struct datdur_t

  "Time since" as a structure: a duration or a date/time
- struct date_t

  Date structure: a date in our time.
- struct dst_rule_year_t

  Rule structure: DST rules for a given (set of) year(s)
- struct timer_t

  The timer data type resp.

Defines

- #define DEFAULT_START_TIME

  Reset (or default) start time.
- #define MARCH_2008_NTP 3413318400

  March 2008 as NTP seconds.
- #define MARCH_2008_UNIX 1204329600

  March 2008 as UNIX seconds.

- #define ms_TIMER_TYPE

  A milliseconds based timer.
- #define MSEC_DAY 86400000

  Milliseconds per day.
- #define MSEC_HOUR 3600000

  Milliseconds per hour.
- #define MSEC_MINUTE 60000

  Milliseconds per minute.
- #define MSEC_WEEK

  Milliseconds per week.
- #define OFFSET2NTPTIME

  same as MARCH_2008_NTP
- #define OFFSET2UNIXTIME

  same as MARCH_2008_UNIX
- #define sec_dat_TIMER_TYPE

  A date / time oriented timer with seconds resolution.
- #define sec_dur_TIMER_TYPE

  A duration / period oriented timer with seconds resolution.
- #define SECONDS_IN_DAY 86400

  Seconds in a day.
- #define SECONDS_IN_WEEK

  Seconds in a week.
- #define SECONDS_IN_YEAR

  Seconds in a (normal) year.
- #define secTime308Loc(x)

  Seconds since March 1st 2008 local time.
- #define startTime308UTC

  The reset / start time in seconds since March 1st 2008 UTC.
- #define TIMER_LAPSED 0x00

  Timer lapsed.
- #define TIMER_RUNNING

  Timer running.
- #define TIMER_STATE_MASK

  upper 4 bits of timer.state (the state)
- #define TIMER_STOPPED

  Timer stopped.
- #define TIMER_TYPE_DEFAULT

  Set type to default or leave untouched.
- #define TIMER_TYPE_FREE

  Timer free.
- #define TIMER_TYPE_MASK

  lower 4 (2) bits of timer.state (the type)
- #define TIMER_UNUSED

  Timer unused.
- #define timerLapsed(timer)

  Is a timer lapsed.
- #define timerRunning(timer)

  Is a timer running.
- #define zoneOffsetSec

  The time zone offset in seconds.
- #define zoneOffsetZone

zoneOffsetSec's byte [1]

Date calculation defines

- #define DAYS_IN_YEAR
  Days in a year.
- #define DAYS_IN4YEARS
  Days in four years.
- #define APRIL_OFF
  Offset of April.
- #define MAY_OFF
  Offset of May.
- #define JUNE_OFF
  Offset of June.
- #define JULY_OFF
  Offset of July.
- #define AUGUST_OFF
  Offset of August.
- #define SEPTEMBER_OFF
  Offset of September.
- #define OCTOBER_OFF
  Offset of October.
- #define NOVEMBER_OFF
  Offset of November.
- #define DECEMBER_OFF
  Offset of December.
- #define JANUARY_OFF
  Offset of (next) January.
- #define FEBRUARY_OFF
  Offset of (next) February.
- #define MARCH_2012
  March 2012.
- #define MARCH_2016
  March 2016.
- #define MARCH_2020
  March 2020.
- #define MARCH_2100
  March 2100.

Functions

- uint32_t datdur2sec (datdur_t ∗timStr)
  Calculate seconds from a datdur_t structure.
- char ∗ formDateIso (char ∗s, date_t ∗tDat)
  Format a date (date_t structure)
- char ∗ formTimDur (char ∗s, datdur_t ∗tRun)
  Format a time (datdur_t structure) as duration.
- char ∗ formTimOfD (char ∗s, datdur_t ∗tTim)
  Format a time (datdur_t structure) as time of day.
- char ∗ formWdShort (char ∗s, uint8_t wd)
  Format the day of week as short clear text.
- void freeTimer (struct timer_t ∗timer)
  Free a timer.
- const dst_rule_year_t ∗ getDSTrule (uint8_t year)
  Get the EU, US &c DST rule data for a given year.
- uint32_t getFATtime (void)

The local time as FAT time.
- void haltTimer (struct timer_t ∗timer)

  Halt a timer.
- uint8_t initTimer (struct timer_t ∗timer, uint32_t period, uint8_t type)

  Initialise a timer.
- uint8_t isEUdstSwitchDay (date_t ∗datStr)

  Is date represented in date structure EU DST switching day.
- uint32_t msClock (void) __attribute__((always_inline))

  Milliseconds since power up.
- uint8_t parseDate (date_t ∗dat, char s[ ], uint8_t si)

  Parse a date.
- uint8_t parseTim (datdur_t ∗tim, char s[ ], uint8_t si)

  Parse a (clock) time.
- void pauseTimer (struct timer_t ∗timer)

  Stop / pause a timer.
- struct timer_t ∗ removeTimerFromList (const struct timer_t ∗timer, struct timer_t ∗list)

  Remove a timer from a list.
- void restartTimer (struct timer_t ∗timer)

  Start a timer with its own interval from now.
- void sec2datdur (datdur_t ∗timStr, uint32_t timSec)

  Convert seconds to a datdur_t structure.
- uint32_t secClock (void) __attribute__((always_inline))

  The system's run time in seconds.
- uint32_t secTime308UTC (void) __attribute__((always_inline))

  Seconds since March 1st 2008 UTC.
- uint32_t secTimeNtpUTC (uint32_t secTime308UTC) __attribute__((always_inline))

  Convert to seconds since January 1st 1990 UTC (NTP time)
- uint32_t secTimeUnixUTC (uint32_t secTime308UTC) __attribute__((always_inline))

  Convert to seconds since January 1st 1970 UTC (Unix time)
- uint8_t setDST (uint8_t dlt)

  Set if current time is DST.
- uint8_t setZoneOffsetSec (uint32_t newOffset)

  Adjust / set zone offset.
- void startNextInterval (struct timer_t ∗timer)

  Start a timer's next interval.
- void startTimer (struct timer_t ∗timer, uint32_t period, uint8_t type)

  Start a timer with interval / type from now.
- void startTimerAbs (struct timer_t ∗timer, uint32_t endTime, uint8_t type)

  Start a timer (with absolute end time)

Date handling functions

- uint8_t setDatByDays (date_t ∗datStr, uint16_t ds)

  Set date structure by days since since March 2008.
- uint16_t getDaysByDat (date_t ∗datStr)

  Get days since since March 2008 by date structure.
- uint8_t getMarchYearByDays (uint16_t ∗daysInYear, uint16_t ds)

  Get year starting March (includes next January and February)

Variables

- uint8_t adjustUTCcount

  Count of secTime308Loc adjustments.
- uint32_t combinedOffset

  The combined offset (local - UTC)
- uint8_t isDST

  Is current time DST.
- uint16_t msAbsClockCount

  The system ms clock value for local time.
- uint16_t msSystClockCount

  The system ms counter / clock value for running / system up time.
- struct timer_t msTimers

  The system milliseconds time and timers.
- struct timer_t secDatTimers

  The system seconds resolution date / time oriented timers.
- struct timer_t secDurTimers

  The system seconds resolution duration oriented timers.
- struct timer_t ∗ timerLists [ ]

  The list of timer lists.
- char ∗ wDaysShort [ ]

  flash array of the (flash) short weekdays

4.35 include/we-aut_sys/uart0.h File Reference

4.35.1 Overview

Serial communication (basics) This file contains the definitions for the basics of weAutSys' serial communication. Please find detailed description of the variables, functions etc. in the modules Serial communication and Serial com. drivers.

This is system software and must not be modified for user or application programs.

This file is part of weAutSys <weinert-automation.de>

Copyright © 2011 Albrecht Weinert, Bochum

Author

Albrecht Weinert <a-weinert.de>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Defines

- #define B1D6MODE_NONE 0

  Usage mode of ports PD6 and PB1: none.
- #define B1D6MODE_RTSCTS_FLOWC 0x11

  Usage mode of ports PD6 and PB1: RTS CTS input flow control.

- #define B1MODE_MASK 0xF0

  Usage mask for PB1.
- #define B1MODE_RTS_FLOWC 0x10

  Usage mode of port PB1 : RTS output flow control.
- #define D6MODE_CTS_FLOWC 0x01

  Usage mode of port PD6 : CTS input flow control.
- #define D6MODE_MASK 0x0F

  Usage mask for PD6.
- #define FROM_BUFFER2UART()

  Driver helper.
- #define isCTSflowC

  Usage mode of port PD6 : CTS input flow control.
- #define isRTSflowC

  Usage mode of port PB1: RTS output flow control.
- #define SER_RECVBUF_EMPTY

  The serial input buffer is empty.
- #define SER_RECVBUF_NOT_EMPTY

  The serial input buffer is not empty.
- #define SER_SENDBUF_EMPTY

  The serial output buffer is empty.
- #define SER_SENDBUF_NOT_EMPTY

  The serial output buffer is not empty.
- #define UART_CAN_SEND

  UART (0) can get send data.
- #define UART_IN_BUF_CAP 127

  The maximum capacity of the serial input buffer.
- #define UART_IN_SPACE_LIM 11

  UART space limit for flow control.
- #define UART_OUT_BUF_CAP 255

  The capacity of the serial output buffer.

Functions

- uint8_t getB1D6mode (void) __attribute__((always_inline))

  Get the usage of the ports PD6 and PB1.
- uint16_t serInBufferd (FILE ∗streams)

  The number of characters buffered from serial input.
- void setB1D6mode (uint8_t mode)

  Set the usage of the ports PD6 and PB1.
- void setUARTflowcontrolByChar (char cC)

  Sets UART flow control by (one) character.
- uint8_t uartClearInBuffer (void)

  Clear the internal buffer for serial input.
- uint8_t uartClearOutBuffer (void)

  Clear the internal buffer for serial output.
- int uartGetChar (FILE ∗stream)

  Get one byte from serial input.
- uint8_t uartGetLine (char ∗line, uint8_t max)

  Read a line from UART.
- uint8_t uartInBufferd (void)

The number of characters buffered from serial input.
- uint8_t uartInErrors (void)

  The accumulated serial input (UART0) errors.
- void uartInit (uint16_t baudDivide, uint8_t x2, uint8_t len, uint8_t parity, uint8_t stopBits)

  Initialise the serial input (UART0)
- uint8_t uartInRetBufferd (void)

  The number of line feeds buffered from serial input.
- uint8_t uartOutSpace (void)

  The buffer space available for serial output.
- uint8_t uartPutBytes (uint8_t ∗src, uint8_t n)

  Put some bytes (characters) to serial output.
- uint8_t uartPutChars (char ∗src, uint8_t n)

  Put some characters to serial output.
- int uartPutCharsP (char const ∗src, uint8_t n)

  Put some characters from program space to serial output.
- uint8_t uartPutSt (char ∗src, const FILE ∗stream)

  Put a RAM string (some characters) to serial output.
- int uartPutSt_P (prog_char ∗src, const FILE ∗stream)

  Put a string (some characters) from flash memory to serial output.
- void uartSetBaudDivide (uint16_t baudDivide) __attribute__((always_inline))

  Set the UART0's baudrate divisor.

Variables

- uint8_t uartInAccErrors

  The serial input's accumulated (or) errors.
- uint8_t uartInBuf [ ]

  The serial input buffer.
- uint8_t uartInBufRetCnt

  The serial input buffer line feeds code counter.
- uint8_t uartInBufRi

  The serial input buffer read/get index.
- uint8_t uartInBufWi

  The serial input buffer write/put index.
- uint8_t uartOutBuf [ ]

  The serial output buffer.
- volatile uint8_t uartOutBufRi

  The serial output buffer read/get index.
- volatile uint8_t uartOutBufWi

  The serial output buffer write/put index.

4.36 include/we-aut_sys/utils.h File Reference

4.36.1 Overview

weAutSys utility / library functions to be used also by application / user software This file contains the definitions for weAutSys' (weAut_01) utility / library functions to be used also by application / user software. This is system software and must not be modified for user or application programs.

This file is part of weAutSys <weinert-automation.de>

Author

Albrecht Weinert <a-weinert.de>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Defines

- #define clearBitMask(bitNum)

  Get the 8 bit "Clear bit mask" for the given bit.
- #define INDEX_OFFSET_LIST2 125

  Offset for token found in second list by searchTokenIn.
- #define setBitMask(bitNum)

  Get the 8 bit "Set bit mask" for the given bit.

Optimised Divide functions

- #define mod2pow(div, po2)

  Modulo by power of 2.
- u8div_t divByVal10 (uint8_t div) __attribute__((pure))

  Divide an unsigned byte by the constant 10.
- u8div_t divWByVal10toByte (uint16_t div) __attribute__((pure))

  Divide an unsigned word by by the constant 10 for a byte quotient.
- u16div_t divWByVal1000 (uint16_t div) __attribute__((pure))

  Divide an unsigned word by by the constant 1000.
- uint8_t mod16byVal7 (uint16_t val) __attribute__((pure))

  Modulo 7 of an unsigned 16 bit value.
- uint8_t mod8byVal7 (uint8_t val) __attribute__((pure))

  Modulo 7 of an unsigned 8 bit value.
- uint32_t div32byVal512 (const uint32_t div) __attribute__((pure))

  Divide unsigned 32 bit by the constant 512.
- uint16_t div16byVal512 (const uint16_t div) __attribute__((pure))

  Divide unsigned 16 bit by the constant 512.
- uint32_t div32byVal1024 (const uint32_t div) __attribute__((pure))

  Divide unsigned 32 bit by the constant 1024.
- uint32_t div32byVal2048 (const uint32_t div) __attribute__((pure))

  Divide unsigned 32 bit by the constant 2048.
- uint32_t div32byVal256 (const uint32_t div) __attribute__((pure))

  Divide unsigned 32 bit by 256.
- uint32_t div32byVal128 (const uint32_t div) __attribute__((pure))

  Divide unsigned 32 bit by the constant 128.

Functions

- uint8_t asDigit (uint8_t c) __attribute__((always_inline))

  Recognise one digit.
- void eightHexs (char ∗begin, uint32_t value)

  Set an eight digit hexadecimal number with leading zeroes into a string.
- void eightHexsBE (char ∗begin, uint32_t value)

  Set an eight digit big endian hexadecimal number with into a string.
- char ∗ form16BytSeq (char ∗s, uint8_t ∗seq)

  Format a sequence of 16 bytes in four hex groups and as characters.
- char ∗ formModTelegr (char ∗s, struct modTelegr_t ∗modTeleg)

  Format a Modbus telegram.
- char ∗ formN20BytSeq (char ∗s, uint8_t ∗seq, uint8_t n)

  Format a sequence of 0..20 bytes in five hex groups and as characters.
- void fourDigs (char ∗begin, uint16_t value)

  Set a four decimal digit number into a string.
- void fourHexs (char ∗begin, uint16_t value)

  Set a four digit hexadecimal number with leading zeroes into a string.
- void fourHexsBE (char ∗begin, uint16_t value)

  Set an four digit big endian hexadecimal big endian number into a string.
- char ∗ getFirstSVNtokenP (char ∗dest, char const ∗src, uint8_t mxLen)

  Copy the first SVN token or a full string from program space.
- uint8_t isContainedIn (char s[ ], const uint8_t c)

  Check if a character is contained in a RAM string.
- uint8_t isContainedIn_P (char s[ ], const uint8_t c)

  Check if a character is contained in a flash string.
- uint8_t parse2hex (uint8_t ∗res, char s[ ], uint8_t si)

  Parse a two digit hexadecimal number.
- uint8_t parseByteNum (uint8_t ∗res, char s[ ], uint8_t si)

  Parse a one byte number.
- uint8_t parseDWordNum (uint32_t ∗res, char s[ ], uint8_t si)

  Parse a four byte number.
- uint8_t parseWordNum (uint16_t ∗res, char s[ ], uint8_t si)

  Parse a two byte number.
- uint8_t searchFirstToken (char s[ ], uint8_t si, uint8_t len)

  Search the first token in a short (RAM) string.
- uint8_t searchTokenEnd (char s[ ], uint8_t si, uint8_t len)

  Search the the end of a token in a short (RAM) string.
- uint8_t searchTokenIn (char s[ ], uint8_t si, uint8_t se, char const ∗const ∗list, char const ∗const ∗list2)

  Match a (short) token in a (RAM) string to a list of flash strings.
- uint8_t searchTokenStart (char s[ ], uint8_t si, uint8_t len)

  Search the next token in a short (RAM) string.
- void threeDigs (char ∗begin, uint16_t value)

  Set a three decimal digit number with leading zeroes into a string.
- void threeDigsB (char ∗begin, uint8_t value)

  Set a three decimal digit number with leading zeroes into a string.
- void twoDigs (char ∗begin, uint8_t value)

  Set a two decimal digit number with leading zeroes into a string.
- void twoHexs (char ∗s, uint8_t value) __attribute__((always_inline))

  Set a two hexadecimal digit number with leading zeroes into a string.

Optimised Multiplication functions

- uint32_t mul16with8 (const uint16_t f16, const uint8_t f8) __attribute__((pure))
  Multiply unsigned 16 bit with 8 bit.
- uint32_t mul16 (const uint16_t ab, const uint16_t cd) __attribute__((pure))
  Multiply unsigned 16 bit with 16 bit.
- uint32_t mul16with17 (const uint16_t ab, const uint16_t cd) __attribute__((pure))
  Multiply unsigned 16 bit with 17 bit.
- uint32_t mul32withVal512 (const uint32_t fac) __attribute__((pure))
  Multiply unsigned 32 bit with the constant 512.
- uint16_t **mul16withVal512** (const uint16_t fac) __attribute__((pure))

Optimised Compare functions

- uint8_t geU32ModAr (uint32_t a, uint32_t b) __attribute__((always_inline))
  A 32 bit unsigned greater equal (ge) comparison for modulo arithmetic.
- uint8_t leU32ModAr (uint32_t a, uint32_t b) __attribute__((always_inline))
  A 32 bit unsigned less or equal (le) comparison for modulo arithmetic.

Endianess Handling functions

- void toggle32endian (ucnt32_t ∗value)
  Toggle the endianess of a 32 bit value.
- void toggle16endian (ucnt16_t ∗value)
  Toggle the endianess of a 16 bit value.
- uint16_t convert16endian (uint16_t value)
  Convert the endianess of a 16 bit value.
- uint32_t convert32endian (uint32_t value)
  Convert the endianess of a 32 bit value.
- void add16littleTo32bigEndian (ucnt32_t ∗opRes, uint16_t op2)
  Add a normal (little endian) 16 bit value to 32 bit big endian.

Variables

- uint8_t const clearBitMasks [8]
  Clear bit masks (table in flash memory)
- uint8_t const setBitMasks [8]
  Set bit masks (table in flash memory)

4.37 main.c File Reference

4.37.1 Overview

The weAutSys based user / application software. The file `main.c` is the place where user software based on weAutSys is put.

User respectively application software is to be provided in the form of protothreads and cycles.

A mandatory protothread function is the applInitThreadF(). It is the place to adapt to the user's board settings and variants as well as to initialise all accordingly. applInitThreadF() is scheduled at the start up phases.

Note

Contrary to its name this `main.c` file must **not** provide the main() function.
The main() function is weAutSys system software.

Author

Albrecht Weinert <a-weinert.de>

Revision:

14

Date:

2015-08-04 15:16:04 +0200 (Di, 04 Aug 2015)

Enumerations

- enum autDemo_t

  The demo applications.

Functions

- ptfnct_t app100msThreadF (struct mThr_data_t ∗uthr_data)

  The user / application specific 100 ms thread.
- ptfnct_t app10msThreadF (struct mThr_data_t ∗uthr_data)

  The user / application specific 10 ms thread.
- ptfnct_t app1sThreadF (struct mThr_data_t ∗uthr_data)

  The user / application specific one second thread.
- ptfnct_t appCliThreadF (struct cliThr_data_t ∗cliThread)

  The user / application specific command line interpreter thread.
- ptfnct_t appInitThreadF (struct pt ∗pt)

  The user / application specific initialisation thread.
- ptfnct_t appSerInpThreadF (struct thr_data_t ∗serInpThread)

  The user / application serial input processing thread.

Variables

- enum autDemo_t autDemo

  The actual demo application.

4.37.2 Enumeration Type Documentation

4.37.2.1 enum autDemo_t

The demo applications.

The exemplary demo applications are mainly for the digital outputs (DO):

- count from 0 to a certain number and then turn the outputs off. Restart in next (n ∗ 10ms) cycle.
- make a pseudo random sequence output. In high speed good for up to eight disco lights.
- use the upper for outputs for a four phase unipolar stepper motor. Besides selecting speed the variants are forward, backwards and back and forth one complete phase (weeper).
- make a stepper demo with random speed and direction.

The (inverse) speed (var autDemoTime etc.) can be selected as a factor for (count down within) the application 10 ms cycle. The default 44 hence is one step every 440 ms.

4.37.3 Variable Documentation

4.37.3.1 enum **autDemo_t autDemo**

The actual demo application.

default: randStep

Examples:

main.c.

4.38 pt/lc-addrlabels.h File Reference

4.38.1 Overview

Implementation of local continuations based on GCC's feature "Labels as values".

Author

Adam Dunkels `adam@sics.se`

This implementation of local continuations is based on a special feature of the GCC C compiler called "labels as values". This feature allows assigning pointers with the address of the code corresponding to a particular C label.

For ATmegas with more than 128K (Harvard) flash memory this feature is inseparably bound to the screwy trampolin implementation. That works but should always be checked with deep distrust when code grows / goes beyond 64K words / 128K bytes (on an ATmega2560 e.g.).

Modifications by A. Weinert (c) 2011

For more information, see the GCC documentation: `http://gcc.gnu.org/onlinedocs/gcc/-Labels-as-Values.html`

Revision:

13

Date:

2014-11-20 17:39:20 +0100 (Do, 20 Nov 2014)

4.39 pt/lc.h File Reference

4.39.1 Overview

Local continuations.

Author

Adam Dunkels `adam@sics.se`

Modification for weAutSys and GCC by A. Weinert

Copyright (c) 2011 Albrecht Weinert

weinert - automation, Bochum

weinert-automation.de weAut.de a-weinert.de

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

4.40 pt/pt.h File Reference

4.40.1 Overview

Protothreads implementation.

Author

Adam Dunkels `adam@sics.se`

Modified by: Albrecht Weinert <a-weinert.de> Modifications' Copyright (c) 2014 Albrecht Weinert, Bochum

This file is part of the weAutSys runtime system.

The changes Adam Dunkel's original are

1) changing the thread function's return type (centrally) to `uint8_t` for GCC C. (It was hard-coded `char`.)

2) unconditionally taking lc-adresslabes.h in lc.h for sake of optimised GCC C and tool usage. (It gives a warning on non GCC. In that case change the the implementation of local continuations.)

3) introducing an alias type for the Protothreads (raw) datastructure.

4) a handful of minor improvements and additions

5) extending the (doxygen) comments

Remark: As the Protothreads return types and the (raw) datastructure are intentionally more than lean. A.W. could also hardly resist the temptation to make a pointer to the thread function, a run flag, the current state and else minimal requirements part of struct pt. This would have brought together what belongs together.

But such (semantic) change would have spoiled the documented handling of Protothreads (as e.g. in psocks). So this was not done here. The necessities of weAutSys and akin were implemented in mThr_data_t (in syst_threads.-h).

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

Data Structures

- struct pt

  A protothread's (raw) data structure.

Defines

- #define PT_ENDED

  thread has finished
- #define PT_EXITED

  thread has finished

- #define PT_WAITING 0

  thread paused due to a condition
- #define PT_YIELDED

  Thread paused.
- #define ptfnct_t

  The return type of a protothread function.

Initialization

- #define PT_INIT(pt)

  Initialise a protothread.

Declaration and definition

- #define PT_BEGIN(pt)

  Declare the start of a protothread inside the protothread function.
- #define PT_END(pt)

  Declare the end of a protothread.

Blocked wait

- #define PT_WAIT_UNTIL(pt, condition)

  Block and wait until condition is true.
- #define PT_WAIT_WHILE(pt, cond)

  Block and wait while condition is true.

Hierarchical protothreads

- #define PT_WAIT_THREAD(pt, thread)

  Block and wait until a child protothread completes.
- #define PT_SPAWN(pt, child, thread)

  Spawn a child protothread and wait until it exits.

Exiting and restarting

- #define PT_RESTART(pt)

  Restart the protothread.
- #define PT_EXIT(pt)

  Exit the protothread.
- #define PT_LEAVE(pt)

  End the protothread.

Yielding from a protothread

- #define PT_YIELD(pt)

  Yield from the current protothread.
- #define PT_YIELD_UNTIL(pt, cond)

  Yield from the protothread until a condition occurs.
- #define PT_YIELD_WHILE(pt, condition)

  Yield while a condition is true.
- #define PT_WAIT_ASYIELD_WHILE(pt, condition)

  Wait while a condition is true mimicked as yield.
- #define PT_OR_YIELD_HERE(pt)

  Yield only if a previous conditional wait did not.
- #define PT_OR_YIELD_REENTER()

  Yield only if a previous conditional wait did not and re-enter that.

Typedefs

- typedef struct pt pt_t

  A protothread's (raw) data structure as type.

Chapter 5

Example Documentation

5.1 individEEP.c

weAutSys (weAut_01) template to initialise EEPROM This file contains the basic MAC and default IP settings for an individual weAut_01 device. The compilation result goes to the EEPROM. The EEPROM content is the only non-volatile storage to individualize a device surviving re-programming with complete flash erasure, besides the fact that flash content should be type specific.

The individualization is provided here by a small serial number (0..20). To get the result into a boards EEPROM do something like:

```
 avr-gcc -c -mmcu=atmega1284p -Wl,--section-start=.eeprom=0x0080 -I.  -I
./include individEEP.c -DserNo=1
 avr-objcopy -j .eeprom --change-section-lma .eeprom=128 --no-change-
warnings -O ihex individEEP.o  individEEP.hex
 avrdude -p atmega1284p -c avrisp2  -P com4   -U eeprom:w:individEEP.hex:
i
```

Change settings according to programming interface and needs.

This file is part of weAutSys <weinert-automation.de>

Author

Albrecht Weinert <a-weinert.de>

Revision:

3

Date:

2014-09-18 15:10:08 +0200 (Do, 18 Sep 2014)

```
//  SVN last change by $Author: albrecht $


#include "pt/pt.h"
#include "we-aut_sys/common.h"
#include "we-aut_sys/network.h"
#include "we-aut_sys/persist.h"

#ifndef serNo
#warning No (small) serializion number defined. Using 0.
#  define serNo 0x00
#endif
```

```
/*  The device's basic default configuration individualized for EEPROM
 *
 *  This structure holds basic configuration data specific to the individual
 *  device. These values typically put starting 0x80 in EEPROM.
 *
 *  Hint: The real address used is found by two step indirection in the EEPROM
 *  itself; see persist.h/c.
 *  Change settings according to programming interface and needs.
 */
INEEPROM (conf_data_t defaultTypeConfData) = {
      {0x40, 0x1B, 0x50, 0xCA, 0xFE, serNo}, // MAC

      0xA1,  // magic number

      /* The default IP configuration */
      {IP_ADD(192, 168, 89, 190+serNo), // IP address
       IP_ADD(255, 255, 255,  0), // net mask
       IP_ADD(192, 168, 89,  11), // default router
       IP_ADD(192, 168, 89,  11), // DNS
       IP_ADD(0, 0, 0,  0), // DNS 2
       IP_ADD(192, 168, 89,  2), // NTP
       IP_ADD(192, 168, 89,  3), // NTP 2
       604800, // lease time (one week)
       0,        // set time (unused)
       DNS1_MSK, // consider only DNS set
       DNS1_MSK | DHCP_MSK  // use DHCP and NTP if possible
      }
};
// As of this writing (April 2013) the magic number is A1 and the EEPROM
// configuration address is 0x80=128. This as well as the structure type used
// may change.
INEEPROM(uint8_t endMarker[3]) = {0x77, 0xFF, 0xDD}; // test end marker example
```

5.2 main.c

This source is a test and demo application for weAut_01 and weAutSys. Also, it is a good starting point respectively pattern for the customer's own (productive) software.

Note

The example file displayed is not the "real" C code.
So, for example documentation (Doxygen) comments are stripped off, but replaced by links to the respective documentation.

```
// Copyright (c) 2011, 2014 Albrecht Weinert, Bochum *
// $Id: main.c 14 2015-08-04 13:16:04Z albrecht $

#include "arch/config.h"  // for sake of Eclipse (4.2.x)
#include "we-aut_sys/ll_system.h"
#include "pt/pt.h" // for sake of Eclipse (4.2.x; can't handle nest'd includes)

#include "we-aut_sys/utils.h"
#include "we-aut_sys/timing.h"
#include "we-aut_sys/uart0.h"
#include "we-aut_sys/streams.h"
#include "we-aut_sys/log_streams.h"
#include "we-aut_sys/proc_io.h"
#include "we-aut_sys/syst_threads.h"
#include "we-aut_sys/cli.h"
#include "we-aut_sys/system.h"

#if defined(weAut_00) || defined(weAut_01) // have Ethernet and SMC
# include "we-aut_sys/network.h"
# include "we-aut_sys/modbus.h"
# include "uip/uip.h"

# include "we-aut_sys/smc.h"    // small memory card low level functions
# include "we-aut_sys/smc2fs.h" // small memory card file system adaption
# include "fatFS/ff.h"
# include "fatFS/diskio.h"
#endif  // have Ethernet and SMC (small memory card)

#include <string.h>

/* Hint to programmers:
 * Constant (final) strings should be declared in flash respectively
 * program memory (128K on ATmega1284, e.g.) and not in the RAM (16K on
 * ATmega1284). If put to RAM it would occupy the same amount of program
 * memory anyway.
 * To put something in flash memory use PROGMEM or better INFLASH().
 */

/* Our own greeting
 *
 * This will be output by our exemplary \ref appInitThreadF() in
 * initialisation phase two.
 */
INFLASH(char const helloReset[]) =
              "\n This is weAutSys OS  ... just starting   (reset) \n \n";
//             0 123456789.0123456789.123456789.1234 6789.123456789v 12 3

enum autDemo_t { none, count, discoBunny,
                                stepperF, stepperR, stepperW, randStep };

enum autDemo_t autDemo = randStep;

static uint16_t autDemoTime = 44;
static uint16_t autDemoTimeUse = 33; // for randStep demo variable speed
static uint16_t timCntDemo = 3; // demo applications cycle division counter

static struct timer_t testTimer0; // timer for app10msThreadF


/*  Command + explanation: application demo "disco bunny"
 *
 *  This must be a flash string.
 *
 *  Hint: The first word (discoBunny) is the command to be recogsised. Case
 *  does not matter and abbreviations will be accepted (dIsco). The first
 *  string will be part of the list of commands by command help or
 *  help -application  <br />
 *  The full text (with all \ref FOLLOW_UP) will be displayed as command
 *  specific help by e.g. disco -help
 */
INFLASH(char const comDiscoBun[])
         = "discoBunny [t/10ms] Demo is DO as flashing disco lights   \n"
FOLLOW_UP  "  The eight bit digital output (DO) should be connected   \n"
FOLLOW_UP  "  to eight powerful lights or beams of different colours. \n"
FOLLOW_UP  "  They will be actuated in a pseudo random manor to give  \n"
FOLLOW_UP  "  a disco lightning effect. The optional parameter is the \n"
FOLLOW_UP  "  period in 10ms (1..65000); for the disco bunny holds:   \n"
FOLLOW_UP  "  The faster the better! \n  \n\0";

/* Command + explanation: application demo "counter" */
INFLASH(char const comDemCount[])
         = "countDemo  [t/10ms] Demo is DO (digital outputs) count   \n"
FOLLOW_UP  "  The eight bit digital output (DO) should be connected  \n"
FOLLOW_UP  "  to a row of eight lights or LEDs DO[0] being the right \n"
FOLLOW_UP  "  one. They will display a (binary) count 0..256. Every  \n"
FOLLOW_UP  "  32 steps the DO will be disabled. The optional parameter  \n"
FOLLOW_UP  "  is the period in 10ms (1..65000); hence 100, e.g., will   \n"
FOLLOW_UP  "  count seconds.   \n  \n\0";

/* Command + explanation: application demo "stepper motor" */
INFLASH(char const comStepF[])
         = "stepFDemo  [t/10ms] Demo is DO rotate stepper motor forward \n"
FOLLOW_UP  "  The upper four bits of the eight bit digital output (DO)  \n"
FOLLOW_UP  "  should be connected to a four phase unipolar stepper    \n"
FOLLOW_UP  "  motor (in the sequence of forward rotation). The output \n"
FOLLOW_UP  "  pattern will effect a forward rotation in half phase    \n"
FOLLOW_UP  "  steps. The optional parameter is the period in 10ms     \n"
FOLLOW_UP  "  (1..65000); 31 e.g. will give a good seconds hand drive \n"
FOLLOW_UP  "  on some standard 4-phase unipolar stepper motors.       \n"
      "  \n\0";


INFLASH(char const comStepR[])
         = "stepRDemo  [t/10ms] like stepFDemo  but reversed  \n"
FOLLOW_UP  "  Enter stepFDemo -help for details.    \n  \n\0";

INFLASH(char const comStepW[])
         = "stepWDemo  [t/10ms] like stepF and stepR combined   \n"
FOLLOW_UP  "  The stepper motor will be driven forward and backward   \n"
FOLLOW_UP  "  with the same number of steps resulting in a kind of    \n"
FOLLOW_UP  "  windshield wiper motion. Enter stepFDemo -help for details. \n"
      "  \n\0";

INFLASH(char const comRandSt[])
         = "randStepDemo [t/10ms] stepper demo, random speed and direction  \n"
FOLLOW_UP  "  The stepper motor will be driven forward and backward quite  \n"
FOLLOW_UP  "  randomly. The optional parameter is the starting period. It  \n"
FOLLOW_UP  "  will speed up every second (stepFDemo -help for details).    \n"
      "  \n\0";

/*  Command + explanation: application demo "dirlist" */
INFLASH(char const comTest01[])
         = "test01     test command (may be empty)      \n";

/*  Command + explanation: application demo "testSD"
 *
 *  This is like \ref comStepF but weeper (back and forth).
 */
INFLASH(char const comTestMC[])
         = "testMC    [-opt] memory card test / debug command \n"
FOLLOW_UP  "  The command is experimental.  \n \n\0";

/* User command: set selected inputs to analogue (AI) */
INFLASH(char const comADinputs[])
         = "ADinputs [mask | -off] set/display analogue inputs   \n"
FOLLOW_UP  "  mask is 00 to FF (without 0x) where ones mark the AI \n"
FOLLOW_UP  "  channels. -off is equivalent to 00 turning all ti DI. \n  \n\0";
/*  User command: set  all (8) inputs to digital (DI) */
INFLASH(char const comADoff[]   )
         = "ADoff    set all inputs digital              \n";


/*  The application defined commands (+ their short explanation)
 *
 *  This is a flash array of pointers to flash strings.
 *  Index 1 .. n must point to all n user commands described above giving
 *  them thus both sequence and (case label) numbers. First (0) and last (n+1)
 *  element must point to an application command headline respectively NULL
 *  as in this example.
 */
INFLASH(char const * const userCommands[]) = {
      helpUserCm, // 0 is no command, is acts always just as separator text
      comADinputs, comADoff, // 1 2
      comDiscoBun, comDemCount, // 3 4
      comStepF,  comStepR,  comStepW,  // 5 6 7
      comRandSt,           // 8
      comTest01, comTestMC, // 9 10 (experimental)
      NULL};  // NULL must be here as end marker

INFLASH(char const weAreAt[]) = " + + address ";

/* Bad hack, experimental
 *
 * This is a proven [sic!] way to avoid the stupid AVR linker error,
 * "relocation truncated to fit ...". One has to experiment with
 * a) if or not and b) with a successful size (like 500).
 *
 http://stackoverflow.com/questions/8188849/
      avr-linker-error-relocation-truncated-to-fit
 */
#if defined(arduinoUno)
INFLASH(char const  pad[500]) = { 0 };
#endif

// test field :
extern uint8_t  _end;
// extern uint8_t  __stack;
uint16_t const  ramVarsEnd = (uint16_t)(&_end);
// end test field

/*  The user / application specific initialisation thread
 *
 *  Declared and described in syst_threads.h
 */
ptfnct_t appInitThreadF(struct pt* pt){
   PT_BEGIN(pt);

//---  Phase 1: pre system init  ---------------------------------------

   // Determine the standard streams (choose one of the following four)
   // initStdStreams(&serStreams); // serial out (UART) for stdio streams
   initStdStreams(&bufLogStreams); // bufLog streams for stdio
   //  initStdStreams(&nulStreams); // use nulStreams as standard streams
   //  bufLogSetConsumer(1); // log stream to UART (0): no consumer set

 // anything else to do here (no clocks are initialised yet)?
 // No, so we just yield to weAutSys to prepare  phase 2
   PT_YIELD(pt); // return from initialisation phase 1. Phase 2 enters below.

//--- Phase 2: post system init  ---------------------------------------

   initTestPins(ON, ON); // we use testpin0 & 1 (and not I²C / twoWire )

// serial communication works now if UART is used for that


/* The first greeting by printf or by uartPutCharsP / uartPutChars or by
   sendSerBytes / sendSerBytes_P in case of bootloader integration.
   This part (alone) could be used as Hello world example.
   Serial output works from phase 2 on; printf  and stdout goes to serial.
   printf and consorts is just good for intermediate tests and demos only
   but strongly deprecated for productive / real time applications.
   Better use  uartPutCharsP or sendSerBytes en lieu de printf.  */

  // printf("Hello, I'm weAutSys! Just starting ... \n\n"); // not recommended

   bufLogSetUART();     // use UART (0) as output for buffered log

#if USE_BOOTLOADER >= 2 // use full bootloader integration
   sendSerBytes_P(LOW_ADD(helloReset)); // this bootloader function uses
   sendSerBytes_P(resetCauseText);     // spin waiting. Do not ever use
   sendSerBytes_P(LOW_ADD(bLF2));     // spin waiting later on. (See below.)
#else                    // else not bootloader provided functions available
   uartPutCharsP(helloReset, 42);  // better 42 end in line 53 with 2 lf
   uartPutCharsP(resetCauseText, 26);
   uartPutCharsP(bLF2, 4);
   while (SER_SENDBUF_NOT_EMPTY); // Spin waiting for USART all sent
#endif
   // Do NOT useabove output method (i.e. spin waiting) anywhere else!
   // ... really ! as weAutSys is a non preemptive runtime (using A. Dunkel's
   // Protothreads). Spin waiting for possibly longer than a handful µs is a
   // crime in that context.
   // It may be tolerable (here) in initialisation, before the real time
   // PLC cycles start. So never ever use constructs like above later!

   PT_YIELD(pt); // return from initialisation phase 2. Phase 3 enters below.

//---  Phase 3: post persistence init  ------------------------------------

#if HAVE_ETHERNET
 // Persistent configuration data were read from EEPROM and (partially)
 // applied already. Those settings might be changed here, before the
 // Ethernet driver (ENC28J60) and stack (uIP) will be initialised.

   // setMACaddP(&...); //  set MAC address here, e.g., if appropriate

   curIpConf.useFlags =  DNS1_MSK | // be DNS client (experimental)
          DHCP_MSK | NTP_CLIENT_MSK; // be DHCP and NTP client
#endif // ethernet

   PT_YIELD(pt); // return from initialisation phase 3. Phase 4 enters below.

//---  Phase 4: post Ethernet init  ---------------------------------------

/* Register the user / application specific threads here:
   10ms, 100ms, 1s (PLC cycles) and command line interpreter (CLI)
   Omitted here is an 1ms application cycle / thread.  */

   registerAppThread(&app1sThread, &app1sThreadF);  // we have 1s cycle
   registerAppThread(&app100msThread, &app100msThreadF); // and 100ms

   // one time initialisations for 100 ms application thread
   // initialise and start the timer used there
   initTimer(&testTimer0, 1851, ms_TIMER_TYPE);    // 1,85 s
   restartTimer(&testTimer0);

   registerAppThread(&app10msThread, &app10msThreadF); // have 10 ms cycle, too
   // one time initialisation for 10 ms application thread
   // timCntDemo = autDemoTime;

   registerAppSerInpThread(&appSerInpThreadF); // we handle UART input for HMI

   /* Register the user CLI we have here */
   registerAppCli(&appCliThreadF, userCommands);  // we have an user CLI

/* Use (also) the UART for CLI in and out by initialising the appSerInpThread
 * structure accordingly. Note the serial input thread (appSerInpThreadF)
 * function and the userCommands already registered (just above).
 */
   initAsCLIthread(&appSerInpThread, &serStreams);

#if HAVE_ETHERNET
/*  Initialise the Ethernet interface usage here: */
   // (stupid) IP demo server app  [1234 conflicts with search agent]
   uip_listen(HTONS(1234)); // start listen on port 1234
   // (simple) echo server app
   uip_listen(HTONS(ECHO_PORT)); // start listen on .. server just to have it

   // enable the use of the (system's) Telnet server app
   uip_listen(HTONS(TELNET_PORT));  // start listen on ... server

   registerAppModFun(appModFun);
   uip_listen(HTONS(MODBUS_PORT)); // start listen on ..  server
#endif //  HAVE_ETHERNET

   PT_YIELD(pt); // return from initialisation phase 3. Phase 4 enters below.

  // bufLogHexHex_P(weAreAt, 12, 0, addrHere());
  // logStackS(14);

//---  Phase 4: post ??may be something in the future??  init
   //  in versions w/o ???? this phase won't be scheduled

   PT_YIELD(pt); // we yield not to get into trouble should phase 4 be added
   // at present (no phase 4) we effectively yield in phase 5 once too often


  // bufLogHexHex_P(weAreAt, 12, 1, addrHere());
  // logStackS(14);

//---  Phase 5: reschedule until ended

   // output the system (greeting) info.
   runOutSysInfoThread(pt, appSerInpThread.cliThrData.systCLIpt,
        appSerInpThread.cliThrData.repStreams);   //  may block more than once
   // This is the place to do all necessary checks and further initialisations
   // especially the time consuming ones before the real time online
   // work starts in the application / user threads registered above

   PT_END(pt) // we never come back to this thread except by reset / restart
} // appInitThreadF(struct pt*)

#if HAVE_ETHERNET
/* UDP implementation
 *
 *  We do not implement any extra UDP protocol here. System software
 *  handles NTP, DHCP etc. So this function just does nothing.
 *  And it hardly should ever be called.
 */
void udpAppcall(void){ }


/*  The uIP event function for the application software
 *
 *  We handle port 7 as just echo.
 *
 *  System software handles Telnet (server) port 23.
 *
 *  Otherwise (in this stupid example) we just handle port 1234 opened in
 *  \ref appInitThreadF(). As it is the only one left we do not even care to
 *  check the port.
 */
void uipAppcall(void){
   if(uip_conn->lport == HTONS(ECHO_PORT)   // Echo protocol
         && (uip_newdata() || uip_acked())) {  // and new (or ack'ed) data
      uip_send(uip_appdata, uip_datalen());
      return;
   } // echo protocol & new data only

   // put further user implemented protocols here
   // if(uip_conn->lport == HTONS(myFunnyPort) handleFunny(); // funny server

   // this (most stupid) example at the end of the chain would handle
   // the port 1234 (listened to above).

   // de facto the following would handle those two events for all ports
   // but (n.b.) system handled port / event combinations will never appear
   // here, as won't ports we don't listen to at all.
   // we don't listen too
   if(uip_newdata() || uip_rexmit()) {
      //         0123456789.12 3
      uip_send("weAutSys OK \n", 13);
   }
} // uipAppcall()

#endif // Ethernet


/*  Exemplary application / user serial input thread
 *
 *  Declared and described in syst_threads.h
 *
 *  This exemplary serial input processing thread will just take a line from
 *  UART input buffer if available and forward it to a command line
 *  interpreter (CLI) thread \ref initAsCLIthread "initialised" to use the
 *  UART as output.
 *
 *  This function may be \ref registerAppSerInpThread "registered" as
 *  appSerInpThread for demo (and is here).
 *
 *  \sa   registerSerInpThread
 */
ptfnct_t appSerInpThreadF(struct thr_data_t * serInpThread) {
   PT_BEGIN(&serInpThread->pt);  // standard first (re-) schedule entry point

   YIELD_FOR_BUSY_CLI(serInpThread);  // let running command execution work
   if ( ! serInpThread->flag  // this flag signals input
        || ( (! uartInRetBufferd() ) // no input
              &&  uartInBufferd() <= (UART_IN_BUF_CAP - UART_IN_SPACE_LIM) ))
      {
      serInpThread->flag = 0;      // reset Flag and
      PT_EXIT(&serInpThread->pt); // exit
   } // no (new) input to process

   uint8_t readC = uartGetCmdLine(serInpThread->cliThrData.line,
      LEN_OF_CLITHR_LINE);

   if (readC < 3) PT_EXIT(&serInpThread->pt); // no input line
   setCliLine(&serInpThread->cliThrData, serInpThread->cliThrData.line,
                               readC); // parse line and prepare CLI

   YIELD_FOR_BUSY_CLI(serInpThread); // start command execution (should end)
   PT_END(&serInpThread->pt) // standard end of thread (no semicolon!)
} // appSerInpThreadF(void)


/*  Exemplary application / user CLI thread
 *
 *  Declared and described in syst_threads.h
 *
 *  This exemplary "command line interpreter" (CLI) function would only be
 *  called (respectively scheduled as thread function) for the
 *  \ref userCommands "user commands" defined
 *  and registered above. \ref systemCommands "System command" will be handled
 *  by system software. Especially the system command "help" would display
 *  the user command defined here when told so. <br />
 *  Notice the trick of having the commands and their short help texts
 *  together in the same structure.
 *
 *  The interpreter’s repertoire can easily be extended.
 *
 *  \sa   registerAppSerInpThread
 *  \sa   systemAbort
 *  \sa   uartGetLine
 *  \sa   searchTokenStart
 *  \sa   searchTokenEnd
 *  \sa   searchTokenIn
 */
ptfnct_t appCliThreadF(struct cliThr_data_t * cliThread){
   if ( ! cliThread->commNumb) PT_EXIT(&cliThread->pt); // exit
   // would normally not be called scheduled with no user command
   PT_BEGIN(&cliThread->pt);  // standard begin of thread part

   uint8_t parsedParam = 0;
      switch (cliThread->commNumb) {
         case 1 : // AD inputs
            if (cliThread->optionNumb >= optOffNum
                      && cliThread->optionNumb <= optAbortNum) { // off etc
               goto fallThroughTo2;
            }
            if (cliThread->optionNumb != optNotGivenNum) goto outAIstate;
            parse2hex(&parsedParam, cliThread->line, cliThread->paramStart);
            goto fallThroughTo2; // Eclipse hates missing breaks
            break;              // jumped over for fall through
         fallThroughTo2:       // just to make Eclipse happy
         case  2 : // AD off
            setAIchannels(parsedParam);
         outAIstate:
            if (!cliThread->repStreams) break; // no report streams no output
            // done by sysCLI             PT_YIELD_OUT_SPACE(&cliThread->pt,
        29, cliThread->repStreams);

            //             0123456789.123456789.1
            char outS[] = " * AD inputs  : non  \n \n";
            if (aiChannels) {
               if (aiChannels == 0xFF) {
                  outS[16] = ’a’; outS[17] = outS[18] = ’l’;
               } else {
                  outS[16] = ’0’; outS[17] = ’x’;
```

```
                 twoHexs(&outS[18], aiChannels);
              }
           } else {
              outS[19] = 'e';
           }
           stdPutS(outS);
           break;

        case  8 :  // randomSt
           autDemo = randStep;
           goto case12_13_Parameter; // -->>>v
        case  7 :  // stepper
           autDemo = stepperW;
           goto case12_13_Parameter; // -->>>v
        case  6 :  // stepper
           autDemo = stepperR;
           goto case12_13_Parameter; // -->>>v
        case  5 :  // stepper
           autDemo = stepperF;
           goto case12_13_Parameter; // -->>>v
        case  4 :  //  countDemo
           autDemo = count;
           goto case12_13_Parameter; // -->>>v
        case  3 :  // discoBunny
           autDemo = discoBunny;
        case12_13_Parameter:   //---------<<<v
           if (cliThread->paramStart != optNotGivenNum) {
              parseWordNum(&autDemoTime, cliThread->line,
                                     cliThread->paramStart); // par * 10ms
              if (autDemoTime < 1) autDemoTime  = 1;
              autDemoTimeUse = autDemoTime;
              timCntDemo = 20; // new cycle after 200 ms
             }
           break;

        case 9: // test01 (experimental)
           if (!cliThread->repStreams) break; // no report streams no command

           //             0123456789.123456789.12345678 9
           char outT[] = " * End of bss(RAM): 0x----  \n";
           fourHexs(&outT[22],(uint16_t)(&_end));
           stdPutS(outT);

#if HAVE_SMC
           if (!lockFsWorkFor(SMC_FS_CLIUSE)) break; // others use FS
           // lock unlock file system structure ... but do nothing at present
           unlockFsWorkFrom(SMC_FS_CLIUSE);
#endif  // HAVE_SMC
           break;

        case 10: //  testMC  [-opt] memory card test / debug
           if (!cliThread->repStreams) break; // no report streams no command
#if HAVE_SMC
           if (!lockFsWorkFor(SMC_FS_CLIUSE)) break; // others use FS
#if defined(weAut_00) || defined(weAut_01)
           fsWork.fRes  = f_open(&fsWork.fil, "/test.txt", FA_OPEN_EXISTING |
      FA_READ);
           PT_YIELD_OUT_SPACE(&cliThread->pt, 31, cliThread->repStreams);
           PT_OR_YIELD_REENTER(); // yield anyhow after long fatFS op.

           if (!fsWork.fRes ) {
              uint16_t ii;
              char buffer[17];  // was 17
              fsWork.fRes1 = f_read(&fsWork.fil, buffer, 15, &ii);
              buffer[ii] = '\n';
              buffer[ii+1] = 0;
              stdPutS(buffer);
              PT_YIELD_OUT_SPACE(&cliThread->pt, 31, cliThread->repStreams);
              PT_OR_YIELD_REENTER(); // yield anyhow after long fatFS op.
             }
```

```
            PT_YIELD_OUT_SPACE(&cliThread->pt, 31, cliThread->repStreams);
            PT_OR_YIELD_REENTER(); // yield anyhow after long fatFS op.
            //              0123456789.123456789.123456789t123456789q
            char outR[] =  " * fil.op.cl: op rd cl  \n";

            if (fsWork.fRes) {
               twoHexs(&outR[14], fsWork.fRes);
             } else {
               if (fsWork.fRes1) twoHexs(&outR[17], fsWork.fRes1);
               fsWork.fRes2 = f_close(&fsWork.fil);
               //  PT_YIELD(&cliThread->pt); // yield anyhow after long op.
               if (fsWork.fRes2) twoHexs(&outR[20], fsWork.fRes2);
            }
            stdPutS(outR);
#endif // weAut_01 | weAut_00
            unlockFsWorkFrom(SMC_FS_CLIUSE);
#endif // HAVE_SMC
            break;

         default:
            break;
      } // switch

      PT_END(&cliThread->pt)
} // exemplary CLI application thread


// user periodic threads (Cycles)  common variables
static uint8_t eKeyWasPressed;  // enter key's past
static uint8_t setHigh;  // how DI thresholds were set on last enter pressed

/*  DO (digital output)  output pattern for disco ("bunny") lights  */
INFLASH(uint8_t bunnyMust[]) = {0x7D, 0x86, 0x6D, 0x43,
                              0x6D, 0xF6, 0x6D, 0x03,
                              0xED, 0x06, 0xED, 0x03,
                              0x6D, 0xB6, 0xCD, 0xDB };

/*  DO output pattern for unipolar stepper motor (upper nipple)
 *
 *  These are 8 steps for a 4 coil unipolar stepper on bits 4..7
 *  of the digital output (DO).
 *  In every step 1 pole (coil) or 2 adjacent poles are on.
 */
INFLASH(uint8_t steppMust[]) = {0x18, 0x39, 0x2A, 0x6B,
                              0x44, 0xC5, 0x86, 0x97, // 0..7
                              0x86, 0xC5, 0x44, 0x6B,
                              0x2A, 0x39, 0x18, 0x97, // 8..F
                              0x86, 0xC5, 0x44, 0x2B,
                              0x1A, 0x89, 0x48, 0x27, // 10 .. 17
                              0x16, 0x85, 0x44, 0x2B,
                              0x1A, 0x89, 0x48, 0x27,  // 18 .. 1F

                              0x16, 0x85, 0x44, 0x2B,
                              0x1A, 0x89, 0x48, 0x27,
                              0x16, 0x85, 0x44, 0x2B,
                              0x1A, 0x89, 0x48, 0x27,
                              0x16, 0x85, 0x44, 0x2B,
                              0x4A, 0x29, 0x18, 0x87,
                              0x46, 0x25, 0x14, 0x8B,
                              0x4A, 0x29, 0x18, 0x97 };
/* Exemplary application / user thread  (10ms)
 *
 *  Has been registered as app10msThread in (above) \c appInitThreadF .
 */
ptfnct_t app10msThreadF(struct mThr_data_t * uthr_data) {
   PT_BEGIN(&uthr_data->pt);
   exitNotFlaggedThread(*uthr_data);
   uthr_data->flag= 0; // work will be done, reset flag. Never forget!

   if  (enterKeyPressed()) { // enter key pressed
         eKeyWasPressed =
              upDIthresh4hyst = 0xFF;  // wider hysteresis all 8 DI channels
         upDIthreshForce = 0;
         toggleStatusLedGn();  // wild blink
   } else if (enterKeyReleased()) { // enter key released
         if (eKeyWasPressed) {
            eKeyWasPressed =
              upDIthresh4hyst = 0x00;  // normal hysteresis for all  DI
            if (setHigh) { // toggle low / high threshold for test
               setHigh = upDIthreshForce = 0;
               setStatusLedGn(OFF); //  // off for less sensitive
            } else {
               setHigh = upDIthreshForce = 0xFF;
               setStatusLedGn(ON); // on for more sensitive
            } // toggle high / low threshold for test
         } // was pressed
   }  // enter key  released


   if (! (--timCntDemo)) {  // counting down til 0
      timCntDemo = autDemoTime; // restart down counter (speed setting)
         // setTestPin1(toggle1 = ! toggle1);

         static uint8_t t1cnt;

         if (! doDriverEnabled()) {
            enableDOdriver(ON);
         } else if ( !doDriverOK() ) {
            enableDOdriver(OFF);
         } else {
            ++t1cnt;

            if (autDemo == count) {   // counter (DO) demo
            // Function: Count this timer’s ticks in t1cnt and output this to
            // digital output. At mod 32 = 31 or at DO driver errors disable
            // the digital output driver for one timer tick.
               if ( (t1cnt & 0x1F) == 0x1F) {
                  enableDOdriver(OFF);
               } else {
                  toDOdriver( t1cnt );
               }
            } else  if (autDemo == discoBunny) {  // disco light demo
               toDOdriver(pgm_read_byte(&(bunnyMust[t1cnt & 0x0F])));
            } else  if (autDemo == stepperF) {  // stepper motor forward
               toDOdriver(pgm_read_byte(&(steppMust[t1cnt & 0x07] )));
            } else  if (autDemo == stepperR) { // stepper motor reverse
               toDOdriver(pgm_read_byte(&(steppMust[(t1cnt & 0x07) + 8] )));
            } else  if (autDemo == stepperW) {  // stepper motor back & forth
               toDOdriver(pgm_read_byte(&(steppMust[t1cnt & 0x0F] )));
            } else  if (autDemo == randStep) { // stepper motor random
               toDOdriver(pgm_read_byte(&(steppMust[t1cnt & 0x3F] )));
               timCntDemo = autDemoTimeUse;
            }
         }
   } // counting down til 0

   PT_END(&uthr_data->pt)
} // 10 ms exemplary application thread.


/* Exemplary application / user thread (100ms)
 *
 *  Has been registered as app100msThread (for demo) in (above)
 *  \c appInitThreadF .
 */
ptfnct_t app100msThreadF(struct mThr_data_t * uthr_data) {
   PT_BEGIN(&uthr_data->pt);
   exitNotFlaggedThread(*uthr_data);
   uthr_data->flag= 0; // work will be done, reset flag. Never forget!

   if (timerLapsed(testTimer0)) {  // timer 0 is elapsed
      static uint8_t t0cnt;
      ++t0cnt;
      if (t0cnt & 1) {
         setStatusLedGn(ON);  // blink with timer 0
      } else {
         setStatusLedGn(OFF); // blink with timer 0
      }
      // restartTimer(&testTimer0);  // just from now
      startNextInterval(&testTimer0);  // frequency exact
   } // timer 0 was elapsed

   PT_END(&uthr_data->pt)
} // 100 ms exemplary application thread.

/* Analogue input results hold */
static uint8_t aiResultHold[8] = {0,1,2,3,4,5,6,7}; // Improbable measurement

/* Exemplary application / user thread (1s)
 *
 *  Has been registered as app1sThread (for demo) in (above)
 *  \c appInitThreadF .
 *
 *  This example shows a cycle's subdivision in two cycles of
 *  half the frequency (here odd and even second schedule). Analogue input
 *  (if any channels are set so) is logged on UART every two seconds here.
 */
ptfnct_t app1sThreadF(struct mThr_data_t * uthr_data){
   PT_BEGIN(&uthr_data->pt);
   exitNotFlaggedThread(*uthr_data);
   uthr_data->flag = 0; // work will be done, reset flag. Never forget!

   if (aiChannels) { // output Analogue Channels (AI) if some set (+DI)
      PT_WAIT_ASYIELD_WHILE(&uthr_data->pt, bufLogStreamsOutSpace(NULL) < 122);
       //            01 23456789.123456789v123456789t123456789q12345678 9c
      char line[] = " \n * AI 7..0 : 7:L 6:L 5:L 4:L 3:L 2:L 1:L 0:L  \n";
      uint8_t msk = 0x80;
      char * digOut = line + 43;
      for (uint8_t chan = 7; msk; --chan, msk >>=1, digOut -= 4) {
         if (!(aiChannels & msk)) { // this channel is DI
            if (filDI() & msk) {
               digOut[2] = 'H';
               aiResult[chan] = 0xFF;
            } else {
               aiResult[chan] = 0;
            }
            continue;
         } // this channel is DI
         threeDigsB(digOut, aiResult[chan]);
      } // for
      if (memcmp(aiResultHold, aiResult,  8)) {
         memcpy(aiResultHold, aiResult, 8);
         bufLogTxt(line, 59);
      }
   }  // ADC

   if(!(--autDemoTimeUse)) autDemoTimeUse = autDemoTime;

   // Wait again until its flag set.   Phase odd
   PT_WAIT_UNTIL(&uthr_data->pt,  &uthr_data->flag);
   uthr_data->flag = 0; // work will be done, reset flag. Never forget!


   PT_END(&uthr_data->pt)
} // One second exemplary application thread.

// Other user threads .... --------------------------------------

#if HAVE_MODBUS

union {
   uint8_t v8[16];
   ucnt16_t c16[8];
} poi;

union {
   uint8_t v8[16];
   ucnt16_t c16[8];
} pii;
```

```
/*  Handle Modbus server events
 *
 *  This exemplary implementation does not distinguish the (old fashioned)
 *  Modbus data models. It maps all incoming data to a byte oriented
 *  POI (process output image) and all outgoing data to a PII (process
 *  input image). In this simple example the length of both is 16 and the
 *  addresses are just taken modulo 16. As lengths 1 .. 16 is accepted.
 */
ptfnct_t appModFun(struct modThr_data_t * const m){

   uint8_t wbC = m->inDataByteCnt; // no of incoming bytes (to POO)
   uint8_t rbC = m->outDataByteCnt; // no of outgoing bytes (from PIO)
   const uint8_t dMod = m->dataModel;

   #if DEBUG_MOD >= 3
   //           0123456789.123456789v123456789t12 3
   char tS[] = " ## ModbAppFun m- w - r - *.... \n";
   if (dMod) tS[16] = dMod;
   if (wbC) twoHexs(&tS[19], wbC);
   if (rbC) twoHexs(&tS[23], rbC);
   fourHexs(&tS[27], (uint16_t)m);
   bufLogTxt(tS, 33);
#endif

   if (dMod == COIL_SINGLE) { // special case single coil
      // const uint8_t bitNum = m->inDatTarg.v8[0] & 0x07;
      const uint8_t bytNum =  (uint8_t)(m->inDatTarg.v16 / 8);
      if (*m->inData ) { // set
         pii.v8[bytNum & 0x0F] |= setBitMask(m->inDatTarg.v8[0]);
      } else { // set else clear
         pii.v8[bytNum & 0x0F] &= clearBitMask(m->inDatTarg.v8[0]);
      } //  clear
      return PT_ENDED;
   } // special case single coil

   if (dMod ==HLD_MASK) { // special case mask single register
      uint8_t iInd = m->inDatTarg.v8[0] & 0x0F; // mod 16 start add
      pii.v8[iInd] = // for simple test only we put POI to PII
      poi.v8[iInd] = (poi.v8[iInd] & (* m->inData)) | (* m->inData + 2);
      pii.v8[iInd + 1] = // for simple test only we put POI to PII
      poi.v8[iInd + 1] = (poi.v8[iInd + 1] & (* m->inData + 1)) | (* m->inData
      + 3);


      return PT_ENDED;
   } // special case mask single register


   // as we map all models to one POI respectively PII and accept all
   // addresses as modulo 16 we just have to check lengths
   if (wbC > 16 || rbC > 16  || ( !wbC && !rbC)) {
      m->errorCode = MODB_EXC_DATA; // length is considered as data by Modbus
      return PT_EXITED;
   }

   if (wbC) {  // we must do write (to POI) before read
     uint8_t iInd = m->inDatTarg.v8[0] & 0x0F; // mod 16 start add
     uint8_t * inData = m->inData;
     for(;;){
        pii.v8[iInd] = // for simple test only we put POI to PII
        poi.v8[iInd] = *inData++;
        iInd = (iInd + 1) & 0x0F;
        if(!--wbC) break;
     }
   } // write to POI

   if (rbC) {  // we must do read (from PII) after write
     uint8_t iInd = m->outDatSou.v8[0] & 0x0F; // mod 16 start add
     uint8_t * outData = m->outData;
     for(;;){
        *outData++  = pii.v8[iInd];
        iInd = (iInd + 1) & 0x0F;
        if(!--rbC) break;
      }
   } // read from PII


   return PT_ENDED;
} // appModFun(struct modThr_data_t *)

#endif // Modbus
```