sys\_arch interface for lwIP 0.6++

Author: Adam Dunkels

The operating system emulation layer provides a common interface

between the lwIP code and the underlying operating system kernel. The

general idea is that porting lwIP to new architectures requires only

small changes to a few header files and a new sys\_arch

implementation. It is also possible to do a sys\_arch implementation

that does not rely on any underlying operating system.

The sys\_arch provides semaphores and mailboxes to lwIP. For the full

lwIP functionality, multiple threads support can be implemented in the

sys\_arch, but this is not required for the basic lwIP

functionality. Previous versions of lwIP required the sys\_arch to

implement timer scheduling as well but as of lwIP 0.5 this is

implemented in a higher layer.

In addition to the source file providing the functionality of sys\_arch,

the OS emulation layer must provide several header files defining

macros used throughout lwip. The files required and the macros they

must define are listed below the sys\_arch description.

Semaphores can be either counting or binary - lwIP works with both

kinds. Mailboxes are used for message passing and can be implemented

either as a queue which allows multiple messages to be posted to a

mailbox, or as a rendez-vous point where only one message can be

posted at a time. lwIP works with both kinds, but the former type will

be more efficient. A message in a mailbox is just a pointer, nothing

more.

Semaphores are represented by the type "sys\_sem\_t" which is typedef'd

in the sys\_arch.h file. Mailboxes are equivalently represented by the

type "sys\_mbox\_t". lwIP does not place any restrictions on how

sys\_sem\_t or sys\_mbox\_t are represented internally.

Since lwIP 1.4.0, semaphore and mailbox functions are prototyped in a way that

allows both using pointers or actual OS structures to be used. This way, memory

required for such types can be either allocated in place (globally or on the

stack) or on the heap (allocated internally in the "\*\_new()" functions).

The following functions must be implemented by the sys\_arch:

- void sys\_init(void)

Is called to initialize the sys\_arch layer.

- err\_t sys\_sem\_new(sys\_sem\_t \*sem, u8\_t count)

Creates a new semaphore. The semaphore is allocated to the memory that 'sem'

points to (which can be both a pointer or the actual OS structure).

The "count" argument specifies the initial state of the semaphore (which is

either 0 or 1).

If the semaphore has been created, ERR\_OK should be returned. Returning any

other error will provide a hint what went wrong, but except for assertions,

no real error handling is implemented.

- void sys\_sem\_free(sys\_sem\_t \*sem)

Deallocates a semaphore.

- void sys\_sem\_signal(sys\_sem\_t \*sem)

Signals a semaphore.

- u32\_t sys\_arch\_sem\_wait(sys\_sem\_t \*sem, u32\_t timeout)

Blocks the thread while waiting for the semaphore to be

signaled. If the "timeout" argument is non-zero, the thread should

only be blocked for the specified time (measured in

milliseconds). If the "timeout" argument is zero, the thread should be

blocked until the semaphore is signalled.

If the timeout argument is non-zero, the return value is the number of

milliseconds spent waiting for the semaphore to be signaled. If the

semaphore wasn't signaled within the specified time, the return value is

SYS\_ARCH\_TIMEOUT. If the thread didn't have to wait for the semaphore

(i.e., it was already signaled), the function may return zero.

Notice that lwIP implements a function with a similar name,

sys\_sem\_wait(), that uses the sys\_arch\_sem\_wait() function.

- int sys\_sem\_valid(sys\_sem\_t \*sem)

Returns 1 if the semaphore is valid, 0 if it is not valid.

When using pointers, a simple way is to check the pointer for != NULL.

When directly using OS structures, implementing this may be more complex.

This may also be a define, in which case the function is not prototyped.

- void sys\_sem\_set\_invalid(sys\_sem\_t \*sem)

Invalidate a semaphore so that sys\_sem\_valid() returns 0.

ATTENTION: This does NOT mean that the semaphore shall be deallocated:

sys\_sem\_free() is always called before calling this function!

This may also be a define, in which case the function is not prototyped.

- err\_t sys\_mbox\_new(sys\_mbox\_t \*mbox, int size)

Creates an empty mailbox for maximum "size" elements. Elements stored

in mailboxes are pointers. You have to define macros "\_MBOX\_SIZE"

in your lwipopts.h, or ignore this parameter in your implementation

and use a default size.

If the mailbox has been created, ERR\_OK should be returned. Returning any

other error will provide a hint what went wrong, but except for assertions,

no real error handling is implemented.

- void sys\_mbox\_free(sys\_mbox\_t \*mbox)

Deallocates a mailbox. If there are messages still present in the

mailbox when the mailbox is deallocated, it is an indication of a

programming error in lwIP and the developer should be notified.

- void sys\_mbox\_post(sys\_mbox\_t \*mbox, void \*msg)

Posts the "msg" to the mailbox. This function have to block until

the "msg" is really posted.

- err\_t sys\_mbox\_trypost(sys\_mbox\_t \*mbox, void \*msg)

Try to post the "msg" to the mailbox. Returns ERR\_MEM if this one

is full, else, ERR\_OK if the "msg" is posted.

- u32\_t sys\_arch\_mbox\_fetch(sys\_mbox\_t \*mbox, void \*\*msg, u32\_t timeout)

Blocks the thread until a message arrives in the mailbox, but does

not block the thread longer than "timeout" milliseconds (similar to

the sys\_arch\_sem\_wait() function). If "timeout" is 0, the thread should

be blocked until a message arrives. The "msg" argument is a result

parameter that is set by the function (i.e., by doing "\*msg =

ptr"). The "msg" parameter maybe NULL to indicate that the message

should be dropped.

The return values are the same as for the sys\_arch\_sem\_wait() function:

Number of milliseconds spent waiting or SYS\_ARCH\_TIMEOUT if there was a

timeout.

Note that a function with a similar name, sys\_mbox\_fetch(), is

implemented by lwIP.

- u32\_t sys\_arch\_mbox\_tryfetch(sys\_mbox\_t \*mbox, void \*\*msg)

This is similar to sys\_arch\_mbox\_fetch, however if a message is not

present in the mailbox, it immediately returns with the code

SYS\_MBOX\_EMPTY. On success 0 is returned.

To allow for efficient implementations, this can be defined as a

function-like macro in sys\_arch.h instead of a normal function. For

example, a naive implementation could be:

#define sys\_arch\_mbox\_tryfetch(mbox,msg) \

sys\_arch\_mbox\_fetch(mbox,msg,1)

although this would introduce unnecessary delays.

- int sys\_mbox\_valid(sys\_mbox\_t \*mbox)

Returns 1 if the mailbox is valid, 0 if it is not valid.

When using pointers, a simple way is to check the pointer for != NULL.

When directly using OS structures, implementing this may be more complex.

This may also be a define, in which case the function is not prototyped.

- void sys\_mbox\_set\_invalid(sys\_mbox\_t \*mbox)

Invalidate a mailbox so that sys\_mbox\_valid() returns 0.

ATTENTION: This does NOT mean that the mailbox shall be deallocated:

sys\_mbox\_free() is always called before calling this function!

This may also be a define, in which case the function is not prototyped.

If threads are supported by the underlying operating system and if

such functionality is needed in lwIP, the following function will have

to be implemented as well:

- sys\_thread\_t sys\_thread\_new(char \*name, void (\* thread)(void \*arg), void \*arg, int stacksize, int prio)

Starts a new thread named "name" with priority "prio" that will begin its

execution in the function "thread()". The "arg" argument will be passed as an

argument to the thread() function. The stack size to used for this thread is

the "stacksize" parameter. The id of the new thread is returned. Both the id

and the priority are system dependent.

- sys\_prot\_t sys\_arch\_protect(void)

This optional function does a "fast" critical region protection and returns

the previous protection level. This function is only called during very short

critical regions. An embedded system which supports ISR-based drivers might

want to implement this function by disabling interrupts. Task-based systems

might want to implement this by using a mutex or disabling tasking. This

function should support recursive calls from the same task or interrupt. In

other words, sys\_arch\_protect() could be called while already protected. In

that case the return value indicates that it is already protected.

sys\_arch\_protect() is only required if your port is supporting an operating

system.

- void sys\_arch\_unprotect(sys\_prot\_t pval)

This optional function does a "fast" set of critical region protection to the

value specified by pval. See the documentation for sys\_arch\_protect() for

more information. This function is only required if your port is supporting

an operating system.

For some configurations, you also need:

- u32\_t sys\_now(void)

This optional function returns the current time in milliseconds (don't care

for wraparound, this is only used for time diffs).

Not implementing this function means you cannot use some modules (e.g. TCP

timestamps, internal timeouts for NO\_SYS==1).

Note:

Be carefull with using mem\_malloc() in sys\_arch. When malloc() refers to

mem\_malloc() you can run into a circular function call problem. In mem.c

mem\_init() tries to allcate a semaphore using mem\_malloc, which of course

can't be performed when sys\_arch uses mem\_malloc.

-------------------------------------------------------------------------------

Additional files required for the "OS support" emulation layer:

-------------------------------------------------------------------------------

cc.h - Architecture environment, some compiler specific, some

environment specific (probably should move env stuff

to sys\_arch.h.)

Typedefs for the types used by lwip -

u8\_t, s8\_t, u16\_t, s16\_t, u32\_t, s32\_t, mem\_ptr\_t

Compiler hints for packing lwip's structures -

PACK\_STRUCT\_FIELD(x)

PACK\_STRUCT\_STRUCT

PACK\_STRUCT\_BEGIN

PACK\_STRUCT\_END

Platform specific diagnostic output -

LWIP\_PLATFORM\_DIAG(x) - non-fatal, print a message.

LWIP\_PLATFORM\_ASSERT(x) - fatal, print message and abandon execution.

Portability defines for printf formatters:

U16\_F, S16\_F, X16\_F, U32\_F, S32\_F, X32\_F, SZT\_F

"lightweight" synchronization mechanisms -

SYS\_ARCH\_DECL\_PROTECT(x) - declare a protection state variable.

SYS\_ARCH\_PROTECT(x) - enter protection mode.

SYS\_ARCH\_UNPROTECT(x) - leave protection mode.

If the compiler does not provide memset() this file must include a

definition of it, or include a file which defines it.

This file must either include a system-local <errno.h> which defines

the standard \*nix error codes, or it should #define LWIP\_PROVIDE\_ERRNO

to make lwip/arch.h define the codes which are used throughout.

perf.h - Architecture specific performance measurement.

Measurement calls made throughout lwip, these can be defined to nothing.

PERF\_START - start measuring something.

PERF\_STOP(x) - stop measuring something, and record the result.

sys\_arch.h - Tied to sys\_arch.c

Arch dependent types for the following objects:

sys\_sem\_t, sys\_mbox\_t, sys\_thread\_t,

And, optionally:

sys\_prot\_t

Defines to set vars of sys\_mbox\_t and sys\_sem\_t to NULL.

SYS\_MBOX\_NULL NULL

SYS\_SEM\_NULL NULL