Pawel Gburzynski

Storage Operations: EEPROM, Flash, SD cards

Version 0.84
April 2009

Introduction

This document describes the available interfaces to external storage. Based on the form of the interface, there are four different kinds of such external storage:

The so-called ***information flash*** (available on MSP430). This is a small chunk of non-volatile on-chip memory (part of the code flash), where the praxis may store some parameters intended to survive a reset.

External ***flash*** or ***EEPROM*** module. This is a non-volatile (typically large) memory area with special access operations. The semantics of some of those operations may differ slightly depending on hardware parameters.

An exchangeable and optional ***SD card***. Such a card appears similar to a flash or EEPROM module, but is accessed with a special (dedicated) set of operations, which, in particular, makes it possible for the card and EEPROM/flash modules to coexist.

External ***SRAM*** or ***SDRAM*** on eCOG. This kind of memory can be made visible as part of the standard address space, but (because of its potentially large size), special operations are required to access those blocks that cannot fit into the address space.

Information flash

You have to put this:

```
#define INFO_FLASH 1
```

into the praxis options file to make the operations on information flash accessible to the praxis.

Conceptually, the information flash is organized into blocks (also called pages), whose parameters are described by the following constants:

`IFLASH_SIZE`	the total size of the information flash in 16-bit words
`IF_PAGE_SIZE`	the size (in words) of one page (note that all pages have to be the same size)

The present parameters on all versions of MSP430 are **`IFLASH_SIZE`** = 128, **`IF_PAGE_SIZE`** = 64, i.e., two pages, 128 bytes each.

The relevance of pages comes into play when you want to erase (initialize) the information flash. Note that, generally, to write something new into the flash, you first have to erase the previous content, and there is no way to erase less that one page.

As the information flash is part of the code area, it can be referenced directly (in the same way as a constant). This (conceptual) declaration is available to the praxis:

```
word IFLASH [IFLASH_SIZE];
```

which means that the praxis can just read the words from the **`IFLASH`** array.

These two operations can be used to modify the contents of this array (which otherwise appears as a constant):

```
void if_erase (int a);
```

The operation erases the page containing the word number **a**. The flash is assumed to be an array of words indexed by integer numbers from 0 to **IFLASH_SIZE** – 1. If **a** is less than zero, then the entire information flash is erased. A system error is triggered, if the specified word number is positive and exceeds the flash size. When the flash is erased, all its words are set to **0xffff**.

```
int if_write (word a, word val);
```

The function stores the specified value (**val**) in the word number **a**. A system error is triggered if **a** exceeds the flash size. The returned value is 0 upon success and **ERROR** (-1), if the read back content of the written location does not match the written value. This usually means that the written location was not erased before write.

If **INFO_FLASH** has been defined as something > 1, e.g.,

```
#define INFO_FLASH 2
```

the following two extra operations become available:

```
void cf_write (address a, word val);
void cf_erase (address a);
```

They provide simple, unconstrained, and unchecked access to the code flash whereby you can write and erase arbitrary areas. The first argument should be a word address pointing into the code flash. The first function attempts to write a word at the specified address, while the second one erases the entire 512-byte flash segment containing the specified location. The functions may misbehave if the specified address points outside the code flash, including the case when it points into the information flash.

External flash or EEPROM

This interface is configured into the system by selecting one of the implemented "STORAGE" modules, e.g., **STORAGE_M95XXX**, **STORAGE_MT29XXX**, **STORAGE_AT45XXX**. Detailed parameters for those modules (corresponding to their specific versions and pin assignments) are provided in file **board_storage.h** in the board's configuration directory.

Here is the list of operations:

```
word ee_open (void);
```

The function initializes the storage for subsequent access. This may involve powering the module up, for example. The returned value is 0 on success and nonzero (typically 1) on failure. Generally, **ee_open** is idempotent, which means that it can be called multiple times with no detrimental effect.

```
void ee_close (void);
```

The function closes the module (which may involve powering it down). It forces **ee_sync** (see below) to flush any pending buffers.

```
word ee_read (lword a, byte *buf, word len);
```

The function reads **len** bytes from the external storage starting at address (byte number) **a** into the provided buffer **buf**. The returned value is 0 upon success, or nonzero (typically 1) on failure. The only legitimate reason for failure is when a portion of the chunk to be read lies outside the available storage area.

```
word ee_write (word st, lword a, const byte *buf, word len);
```

The function writes **len** bytes from **buf** to the external storage starting at address (byte number) **a**. The returned value is 0 upon success, or nonzero (typically 1) on failure. The only legitimate reason for failure is an attempt to write outside the storage area.

The issuing process can prepare itself for the possibility that the operation may occasionally take a somewhat less than trivial amount of time. The exact circumstances depend on the module, and typically involve situations like buffer boundaries, forced erase, and so on. If the first argument is not **WNONE**, then it identifies a state where the invoking process wants to be resumed to restart the operation. In such a case, the driver may force **release** for the invoking process indicating that it should be resumed in the specified state when the operation has a new chance for progress. If the first argument is **WNONE**, the operation will hold the CPU until the transaction is complete (regardless of the circumstances).

Note that the proper usage of the state argument is to re-execute the operation, e.g.:

```
...
entry (EE_WRITE)
      st = ee_write (EE_WRITE, a, buf, buflen);
      // write complete
...
```

The area to be written may have to be erased beforehand. This erasure may be implicit and transparent, e.g., as in the case of AT45xxx (from Atmel), but an understanding of what actually happens when data is written may help the programmer improve the praxis's performance. Continuous writing is usually preferred. For example, in the case of AT45xxx, the written data is put into an internal buffer (present inside the module). When the buffer becomes filled, it is flushed (written) in a special way that pre-erases the respective sector of the actual storage. Two such buffers are available, which means that the praxis can efficiently write at two different places concurrently. With the present implementation of the driver, a buffer is also required for reading, so reading may disrupt writing and cause buffer thrashing. This does not affect the formal semantics of writing: when a buffer is needed for reading, its previous contents are flushed, and the buffer is assigned the new role. Later, when the write operation resumes in the same area, the buffer will be reloaded with the image of the partially written sector. While such activity patterns do not affect data integrity, a performance penalty is incurred when they persist.

If the praxis never overwrites once written data, except in well known and rare circumstances, it is possible to pre-erase the target area and then use a faster version of writing that does not erase the sector before flushing a buffer. This will work even if the buffer is flushed and re-read possibly multiple times, as long as the new writes go to areas that have not be written since the last explicit erase. To enable this feature, include this definition:

```
#define EE_NO_ERASE_BEFORE_WRITE 1
```


in the praxis options file. This setting only applies to AT45xxx modules and tells **ee_write** to use a non-erasing variant of the buffer flush operation. It will only work if you erase the storage explicitly before writing (see below), and never write something different twice into the same area.

The need to erase before write may be inherent in a given module type (e.g., MT29xxx), i.e., an explicit erasure may be required before writing. This also means that overwriting once written data is prohibited, unless preceded by erase.

With this function:

```
lword ee_size (Boolean *er, lword *eru);
```

the praxis can learn the size (in bytes) of the external storage (which is returned as the function's value). If any (or both) of the two argument pointers is not **NULL**, the function will return via it some information regarding the storage module. In particular, **er** = **YES** means that explicit erasure is needed before a safe write (i.e., write does not automatically imply erase). The second value (**eru**) returns the size of the minimum erasable chunk in bytes.

This operation erases the specified fragment of the storage:

```
word ee_erase (word st, lword from, lword upto);
```

where **from** is the starting byte number and **upto** is the last byte number of the area to be erased. Similar to **ee_write**, the first argument (if not **WNONE**) indicates a state to resume the operation, if it takes too much time. Note that the erasure of a large area may take several seconds.

Note that the proper usage of the state argument (both for **ee_erase** and **ee_write**) is to re-execute the operation, e.g.:

```
...
entry (EE_CLEAN)
      ee_erase (EE_CLEAN, 0, 0x1ffff);
      // erase complete
...
```

Depending on the physical characteristics of the module, the specified boundary of the area to be erased may be rounded to a multiple of the minimum erasure unit (as returned by **ee_size**). Generally, for a module that requires *no* explicit erasure before write, the specification is honored exactly.

The function returns 0 upon success and nonzero on failure. The only possible reason for failure is when **upto** is less than **from** or **from** is greater than or equal to the storage size. If **upto** exceeds the storage size, it is truncated to point to the last byte. The special case **from** = **upto** = 0 means erase all storage.

```
word ee_sync (word st);
```

Depending on the module's characteristics, this operation flushes any partially filled buffers and, generally, brings the module to a consistent state before a power down. The meaning of the state argument is as for **ee_erase** and **ee_write**. The function always returns zero.

```
void ee_panic (void);
```

This function is not intended to be used by the praxis during normal operation. Its role is to quickly (and in the simplest and most failsafe way) synchronize the storage module with any dirty RAM buffers, usually making it impossible to continue normal operation. It is called by the system upon soft reset, e.g., triggered by a system error, just a moment before the CPU is restarted.

SD card interface

To make sure that the SD interface is compiled in, insert this definition:

```
#define STORAGE_SDCARD  1
```

into the praxis options file. File **board_storage.h** in the board's configuration directory provides implementation-specific parameters (like pin assignments).

The present interface is preliminary and looks similar to the EEPROM interface. Here is the list of functions:

```
word sd_open (void);
```

The function initializes the SD card. It can return one of the following values (defined in **sdcard.h**):

0	Success: the card has been initialized and is ready for subsequent operations.
SDERR_NOMEM	The driver couldn't allocate memory for the buffer.
SDERR_NOCARD	There was no response from the card, which probably means that there is no card in the socket.
SDERR_TIME	Communication timeout, e.g., something wrong with the card (possibly the card is not supported).
SDERR_NOBLK	The driver was unable to set the block size to 512, which hints at an unsupported card.
SDERR_UNSUP	The driver was unable to determine the card's size, which also means that the card cannot be supported.

Any of the last three values means that there is a fundamental problem with the card (it will not work with the present version of the driver).

The driver needs a 512-byte buffer, which is allocated by calling **umalloc**. The buffer remains allocated for as long as the card is in use, and will be returned with **sd_close** (see below).

```
lword sd_size (void);
```

The function returns the size of the card in bytes. Owing to the way information on the card is addressed, the maximum supported card size is a little short of 4GB. A 4GB card or larger will show as having **0xfffffffe0** bytes, i.e., one block away from the wraparound of the **lword**-sized counter. The maximum card size tested so far is 2GB.

```
word sd_read (lword a, byte *buf, word len);
```

The function reads **len** bytes from the card into the buffer **buf** starting at address (byte number) **a**. The possible return values are:

0	Successful completion: the bytes have been read.
SDERR_NOCARD	The interface has not been opened (**sd_open**) or has been closed (with **sd_close**).
SDERR_RANGE	The specified chunk of data exceeds (perhaps only partially) the card's size.
SDERR_NOBLK	Bad response to a block read operation (the card may have been removed).
SDERR_TIME	Response timeout (the card may have been removed).

The present version of the driver does not verify the CRC codes of the read blocks, nor does it append CRC codes to written blocks. Some sources say that this is the recommended mode of operation over the SPI interface. We may add CRC verification later, if needed.

```
word sd_write (lword a, const byte *buf, word len);
```

The function writes **len** bytes from the buffer **buf** to the card starting at address (byte number) **a**. The possible return values are:

0	Successful completion: the bytes have been written.
SDERR_NOCARD	The interface has not been opened (**sd_open**) or has been closed (with **sd_close**).
SDERR_RANGE	The specified chunk of data exceeds (perhaps only partially) the card's size.
SDERR_NOBLK	Bad response to a block read/write operation (the card may have been removed).
SDERR_TIME	Response timeout (the card may have been removed).

Note that, in contrast to **ee_write**, there is no state argument. Writing goes through the in-RAM buffer, which, during continuous write, is only flushed when filled. The same buffer is also used for reading, so reading (from a different area) will disrupt writing and may cause buffer thrashing. This is all transparent except for the possible performance penalty.

Normally, when a write operation refers to a different (than the previous one) 512-byte block, the block is first read from the card. The praxis can be compiled with this declaration:

```
#define SD_NO_REREAD_ON_NEW_BLOCK_WRITE 1
```

in the options file. Then, whenever the write operation causes something to be written into a new (uncached) block, and at the beginning of that block, the driver will not read the block from the card, assuming that the praxis intends to overwrite the entire previous contents of the block. Use with due diligence.

No explicit erase operation is implemented at present (we may add it later). Erase happens automatically when writing, and is completely transparent in all cases.

The driver has an option to automatically put the card into an idle state after every operation – to conserve power. This option is off by default, as it may perceptibly increase the average access time for some cards. If you want to switch it on, insert this declaration:

```
#define SD_KEEP_IDLE 1
```

into the praxis's options file. This function:

```
void sd_idle (void);
```

can be used to explicitly put the card into idle state from the praxis. It ignores any possible errors. There is no need to move the card to the active state before an operation: it happens automatically. The function does not sync the card (there is no need do do that before idling).

```
word sd_sync (void);
```

The function flushes the cached buffer (if dirty) to the card. The returned values are 0 (indicating normal completion), or **SDERR_NOBLK** / **SDERR_TIME**, indicating a problem. The function does nothing if the interface is closed (returning zero).

```
void sd_close (void);
```

The function closes the interface, if it is open, or does nothing otherwise. In the former case, the function also forces **sd_sync**, but it doesn't verify its return code. The 512-byte cache buffer (allocated by **sd_open**) is deallocated. The card is put into the idle (low-power) state.

```
void sd_erase (lword from, lword upto);
```

The function erases the area between **from** and **upto** (interpreted as byte addresses), inclusively. The arguments have essentially the same meaning as their counterparts in **ee_erase**. In particular, zero as the second argument stands for the last byte of the card. Note that, in contrast to **ee_erase**, **sd_erase** has no *state* argument, i.e., the function always waits for the completion of the operation. Usually it is very fast, even when the erased area is large. For example, a 2GB Lexar card is erased completely (by executing **sd_erase (0, 0)**) in less than one second.

Even though the function attempts to make a sensible job in all cases, it is strongly recommended to avoid arguments that do not encompass an entire number of 512-byte blocks. Any initial and/or trailing fragment of a full 512-byte block is erased by explicitly writing zeros into the affected bytes. This need not be consistent with the way full blocks are erased. For some cards, bytes in such blocks end up filled with zeros, for some others they are set to all-ones.

```
void sd_panic (void);
```

This function plays the same role as **ee_panic**, i.e., it provides the most economical and foolproof way to flush any pending contents of the RAM buffer to the card before restarting the system.

Accessing external SRAM/SDRAM on eCOG

An add-on SRAM/SDRAM module for eCOG can be mapped into the standard address space, the only problem being that, usually, it cannot be mapped all at once. Consequently, the module is partitioned into 32KW chunks, and only one of those chunks can be present in the address space at any given time (at location **0x4000**). The idea is to transparently (or as transparently as possible) extend the on-chip RAM by 32KW of SDRAM, while making the remaining portion of the SRAM/SDRAM module accessible via special operations. Thus, the first 32KW block (the zero block) of SDRAM is mapped *permanently* into the standard address space and provides memory for **malloc**. However, the permanence of this mapping is conceptual: the operations for accessing the remaining blocks of SDRAM may temporarily unmap the zero block and map there some other block, but this always happens without releasing the CPU (except to interrupts). Thus, as long as interrupts confine themselves to the on-chip RAM, they are in the clear.

These constants describe the parameters of add-on SRAM/SDRAM:

SDRAM_BLKSIZE	the size of an SRAM/SDRAM block (0x8000)
SDRAM_NBLKS	the total size of SRAM/SDRAM in blocks
SDRAM_SPARE	the number of blocks for special access (**SDRAM_NBLKS**-1)

SDRAM_SPARE gives the number of so-called *spare* SRAM/SDRAM blocks, i.e., ones that cannot be accommodated directly in the standard address space and have to be referenced via special operations. The total size of memory available this way is equal to **SDRAM_SPARE * SDRAM_BLKSIZE** (which is usually a **lword** number).

Two operations are available to reference the spare memory:

```
void ramget (address to, lword from, int nw);
void ramput (lword to, address from, int nw);
```

The first operation transfers **nw** words from the spare area to RAM. The RAM address is determined by the first argument, and the second argument points to the first byte of the chunk in spare SRAM/SDRAM. The first byte in the first spare block has number zero. The second function works in the opposite direction. A system error is triggered if the transferred chunk is outside the bounds of spare SRAM/SDRAM.

If one end of the transfer is in the on-chip RAM, the transfer is implemented by temporarily unmapping the “permanent” SRAM/SDRAM block, mapping in the appropriate spare block, and copying the bytes directly. Otherwise (if the transfer involves the permanent SRAM/SDRAM block), it is carried out by multiple re-mappings and uses a small buffer in on-chip RAM. At the end, the permanent block is brought back into its normal place.

Generally, interrupts use only static data structures (stored in on-chip RAM), so they are not affected by the fact that the SRAM/SDRAM component of the address space is switched back and forth during a spare memory transfer. One problem that must be taken care of is a timer counter stored in the SRAM/SDRAM block (e.g., in a **malloc**'ed structure). The eCOG version of PicOS kernel will signal a system error when an attempt is made to set up a timer counter stored in the SRAM/SDRAM block.