Gerhard Jacobs
Jens Altmann

SDL for ATmega$_{xx}$ Controllers

Contents

SDL for ATmega$_{xx}$ Controllers 1

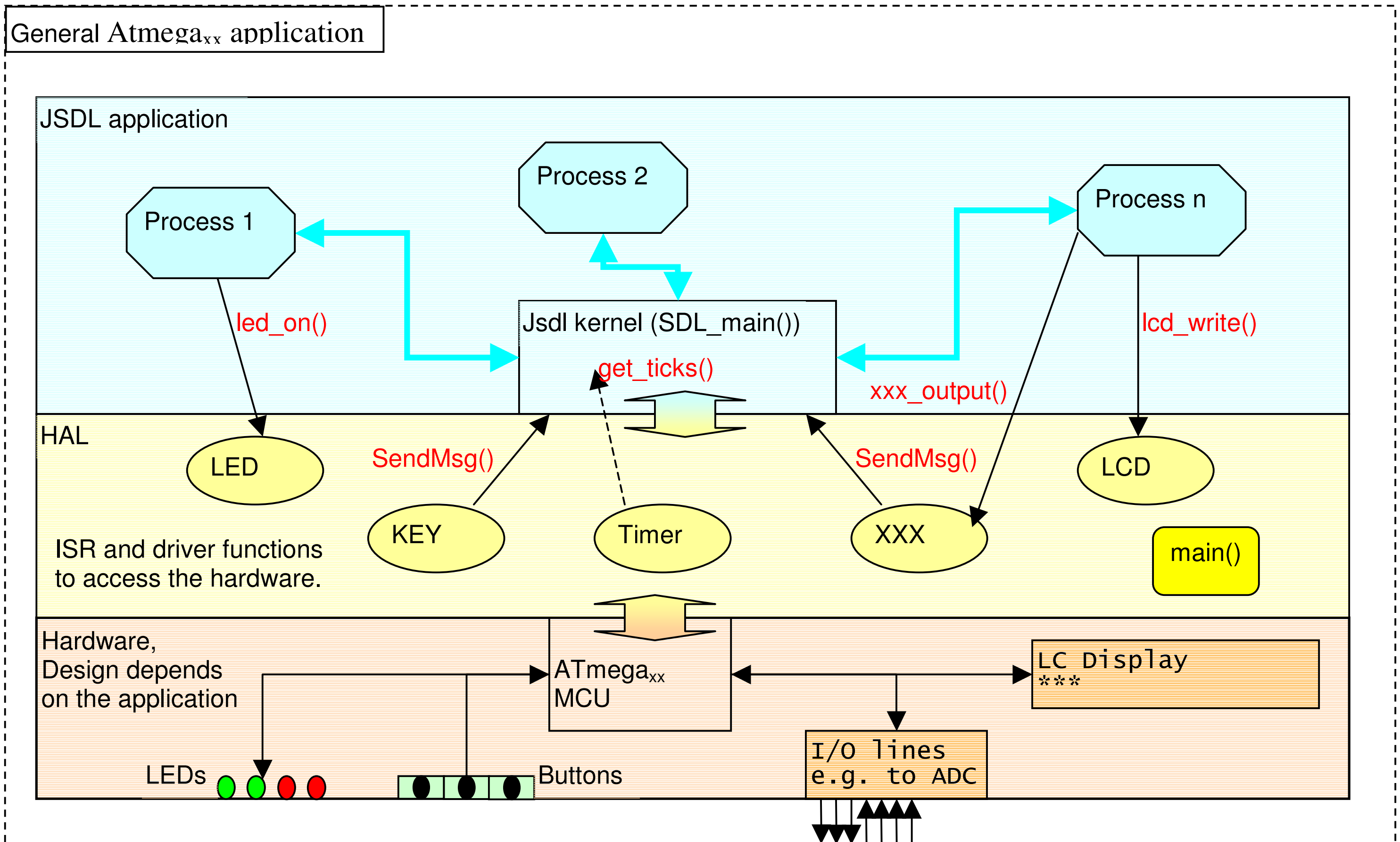
1 ATmega$_{xx}$ SDL Applications 2
1.1 Hardware Abstraction Layer (HAL) 2
1.1.1 LED 3
1.1.2 LCD 3
1.1.3 Timer 3
1.1.4 KEY 3
1.1.5 XXX 3
1.1.6 Generic ISR Interface 4

2 Enhancements to the *JSDL* tool 5
2.1 Accessing the input signal parameters 5
2.2 Convenience macros for use in the SDL code 6
2.3 New code generator options 7
2.4 Use of ‘Declarations’ and ‘Comments’ 9

1 ATmega$_{xx}$ SDL Applications

An Atmega$_{xx}$ SDL application mainly comprises of two parts:

- The JSDL application, (including the JSDL kernel) which can be simulated on a Windows host; in which case the header files of the HAL must provide definitions for the API functions.
- The Hardware Abstraction Layer (HAL) which is application specific and provides an API for accessing the peripheral hardware through the MCU as well as the Interrupt Service Routines (ISR) and initialisations.



1.1 *Hardware Abstraction Layer (HAL)*

The HAL consists of several modules and abstracts the access to the hardware employed by the project, e.g. some LEDs, a keypad or some buttons, a LC Display, etc..

- The HAL provides an API with functions that can be called directly by the SDL processes (or the JSDL kernel) hiding the hardware details from the application. For example, the application can turn on/off a LED without knowing which port it is connected to.
- The HAL also provides Interrupt Service Routines (ISR). The SDL application must register with the ISR telling it which process gets what signal when the interrupt occurs.
- The HAL also implements all the necessary initialisations of the hardware, and the *main()* function. The *main()* function calls *SDL_main()* after all is set up.

1.1.1 LED

This HAL module handles the LED and the buzzer

- Files led.c, led.h
- API

led = {LED1|LED2}
void led_init(); for internal use, initialize the LED port lines
void led_on(led); turn on LED *led*.
void led_off(led); turn off LED *led*.
void led_toggle(led); toggle LED *led* between on and off.
void buzz(int t); sound buzzer for *t* ms..

1.1.2 LCD

This HAL module handles the LCD

- Files lcd.c, lcd.h
- API

void lcd_init(); for internal use, initialize the LCD port lines and the display
void lcd_write(char *s, char l); write the text *s* (max. 16 chars) to line *l* (l = [0..1])
void lcd_putc(char c); write the character *c* to the current curser position.
void lcd_seek(char r, char l); place the cursor to the position.*r* on line *l* (l = [0..1])

1.1.3 Timer

This HAL module handles the timer for the tick of the SDL timers (~10 ms)

- Files app_timer.c, app_timer.h
- API

void timer_init(); for internal use, initialize the timer/counter (16 bit TCNTR1)
unsigned short get_ticks(); for use of the SDL kernel, called by *GetCurTime()*.
SIG_OUTPUT_COMPARE1A(); ISR for the timer device. Internally used.

1.1.4 KEY

This HAL module handles the key connected to the two external interrupt lines (INT0, INT1)

- Files key.c, key.h
- API

void key_init(); for internal use, initialize the input lines and interrupts
SIG_INTERRUPT0(); ISR for the key input lines. Internally used.
SIG_INTERRUPT1(); ISR for the key input lines. Internally used.
void key_signal(ivect_t v, msgID_t m); Function to register SDL signals

1.1.5 XXX

TBD

1.1.6 Generic ISR Interface

Every HAL module, that offers interrupt events (defined as signals from ENV in the SDL system) must provide a generic API function for the SDL processes to register a signal. Only one process can get the interrupt event (signal). The (optional) signal parameter is defined by the HAL module's implementation. A HAL module can support several interrupt sources (represented by their respective interrupt vector number).

- API
  void xxx_signal(ivect_t v, msgID_t m); can be called by any process and registers the signal '*m*' for the interrupt source '*v*', as defined by the respective HAL module, normally the interrupt number). Note that the receiving process is implicit in the signal number! So usually only the receiving process would call this function. This is done best in the start transition of the process.

Example:

key.h:

```
#define IVECT_BUTTON_1 (ivect_t)(0)
#define IVECT_BUTTON_2 (ivect_t)(1)


/* for the signal parameter */
#define KEY_PRESSED 1
#define KEY_RELEASED 2

#ifdef __AVR
extern void key_signal(ivect_t v, msgID_t m);
#else // for host simulation
#define key_signal(v,m)
#endif //__AVR
```

process1.c (generated):

```
#include "key.h"
…
/* the signals are defined in sysinc.h (generated) */
key_signal(IVECT_BUTTON_1, KEY_UP);
key_signal(IVECT_BUTTON_2, KEY_DOWN);
 …
```

2 Enhancements to the *JSDL* tool

Thanks to Jens Altmann (the author of the JSDL tool), who left the source code to me to make these enhancements possible.

The enhancements are mainly aiming at

- Conveniently handling of the signal parameters (and the signal ID) on the SDL diagram level.
- Better accommodating the 8-bit nature of small µControllers by avoiding 32-bit integers and using *char* and *short* whenever possible.
- Correction to the *makefile* generation (for *nmake* being a bit picky).

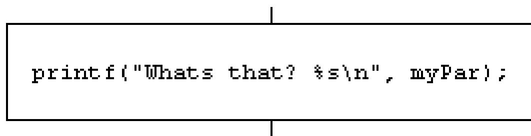
2.1 Accessing the input signal parameters

In general, the single parameter of a signal is always addressed by '`thisProc->pMsg->msg`' (#defined as *`msgpar`* in "*kernel_if.h*"), where '`msg`' is always of type 'unsigned long' or 'unsigned short', depending on the code generation settings. Thus the parameter has to be type casted to whatever the parameter is used for. Of course the use of parameters is limited to types that can be casted from '*long*' (or '*short*') , like *char*, *int* or *struct xyz** (pointers); you could not use *struct xyz* ! ☞ **Caveat**: be careful when sending pointers around ! You must not use pointers to local variables, because they're gone when the sending process or ISR returns ! Accordingly take care with pointers to allocated memory (though this is not an issue for the Atmel controllers), remember to free the memory eventually.

This enhancement implements convenient way to support signal parameters in *input* symbols in the graphical SDL design:

In an ***Input Symbol*** write (telling the code generator you want to access the parameter):

In a ***Statement Symbol*** write (the parameter is used as *char ** here):



Where each occurrence of '***myPar***' (except in the Input Symbol) will result in the source text

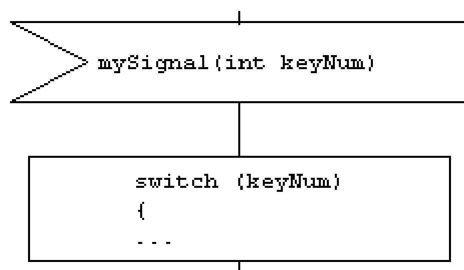
```
((char*) msgpar)
```
[1]

So the *printf* call in the generated code would read:

```
printf("What's that? %s\n", ((char*) msgpar));
```

[1] See ***kernel_if.h*** : `#define msgpar (thisProc->pMsg->msg)`

Another example:



Leads to the generated source text (for the '*Statement*')

```
switch(((int) msgpar)) {
```

...

A pair of parentheses '(' ')' with a type + identifier (like a variable declaration) following an input signal name declares a signal parameter, e.g. `(int myPar)`.
Identical signal parameter definitions may be given in different signals, like
"*Signal1(int keyState)*" and "*Signal2(int keyState)*", but **not** "*S1(**int** keyState)*" and "*S2(**char** keyState)*".

Each occurrence in the program text is expanded to dereference the parameter casted to the defined type, e.g. (`myPar` → `((int) msgpar)`).

This substitution is actually done by the preprocessor by way of macros; the code generator just collects the signal parameters and creates macros for them, like:

```
#define myPar ((int) msgpar)
```

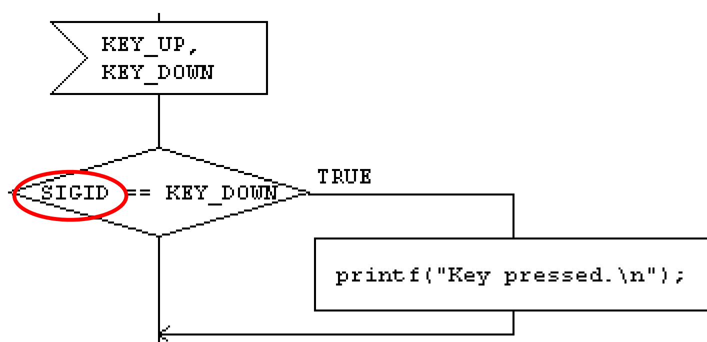
☞ **Caveat**: You must not use the names of the signal parameters for variables somewhere else !

2.2 Convenience macros for use in the SDL code

New macro to access the signal number:

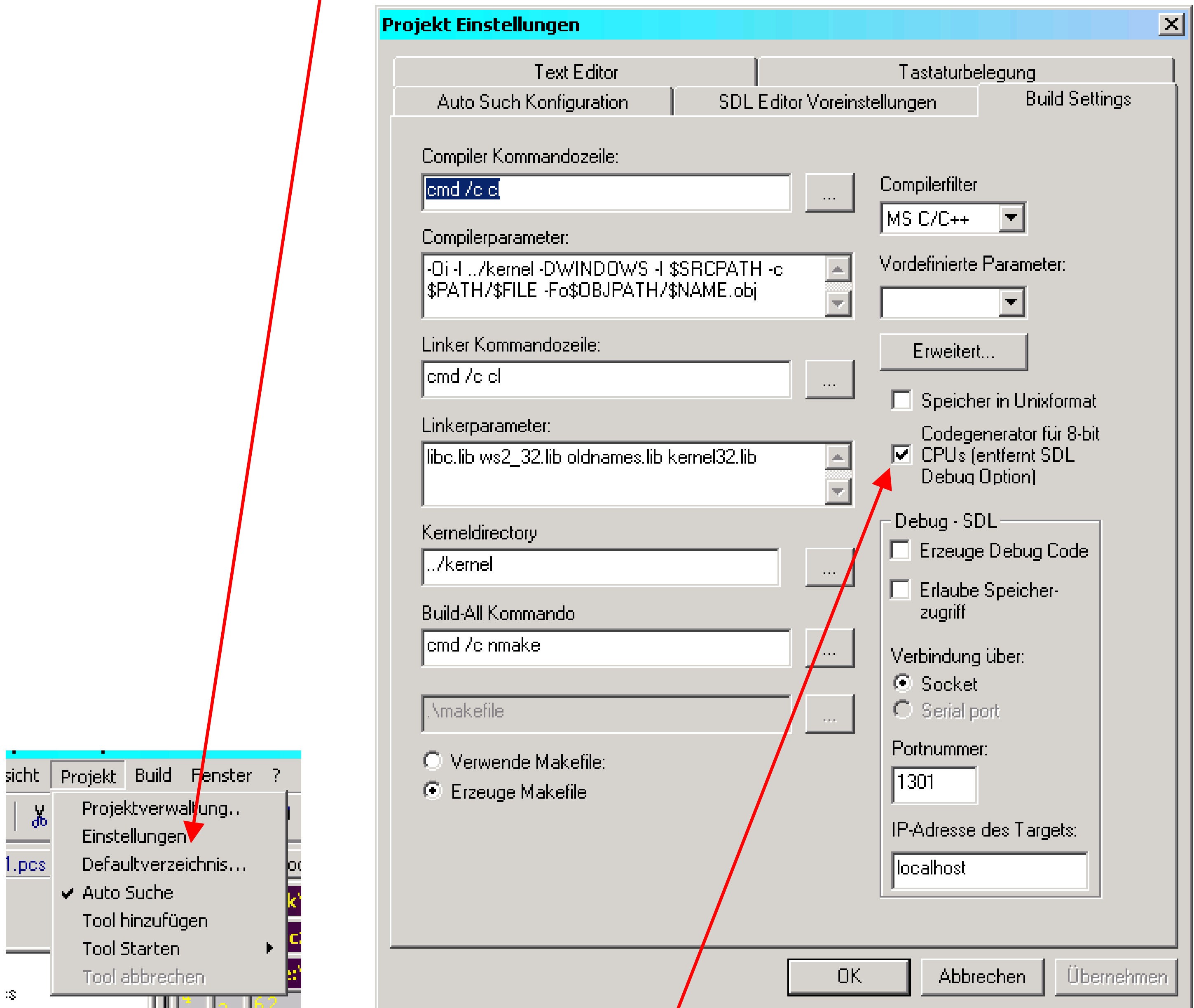
```
#define SIGID (thisProc->pMsg->msgID)
```

Usage (e.g. in a *Decision*):



2.3 New code generator options

The tab '*Build Settings*' in the dialog
from the menu *Projekt* → *Einstellungen*



Gets an additional option:

- Generate code for 8-bit CPU
  The main effect of this option is that the signal numbers are not built as 32-bit constants
  `([receiver process number] << 16) + [16-bit signal number]`
  but rather as 16-bit constants
  `([receiver process number] << 8) + [8-bit signal number]`
  so that the **`msgID`** member in the `msg_t` can be of type ***unsigned short***.

- So for example the generated file "**sysinc.h**" would be with respect to the signals:
  '***normal***' mode:

```
/***** signals **************************************************************/
#define  KEY_UP                                           0x00030007  /*@S@KEY_UP*/
#define  KEY_DOWN                                         0x00030008  /*@S@KEY_DOWN*/
#define  T4                                               0x0000000A  /*@T@T4*/
```

  8-bit CPU mode:

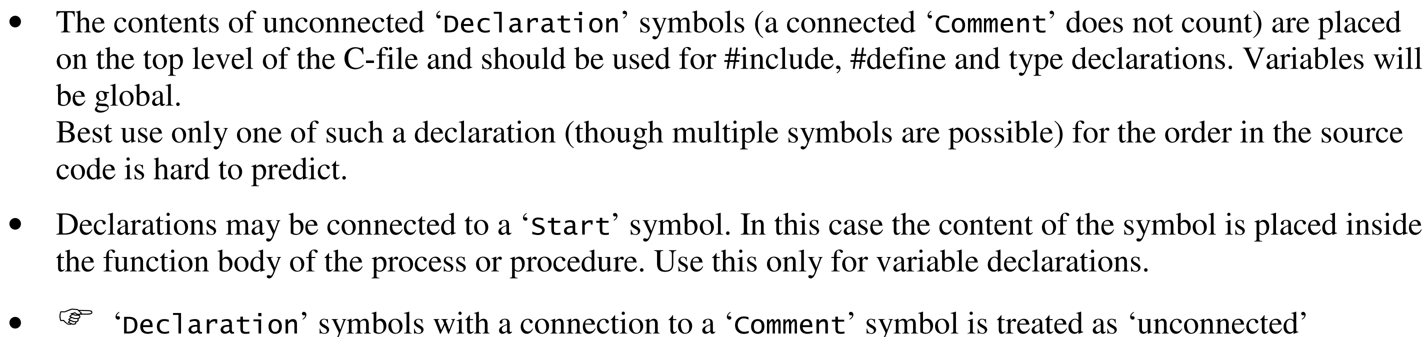
```
/***** signals **************************************************************/
#define  KEY_UP                                           0x0307
#define  KEY_DOWN                                         0x0308
#define  T4                                               0x000A
```

  ☞ **This limits the number of processes and the number of signals + timers to 255, but this should be sufficient for 8-bit controller applications.**

  Also, the preprocessor macro **`_8_BIT_CPU`** is defined in the header file "sysinc.h". If you compile the code with the AVR SDL-kernel for the Atmega$_{xx}$ controllers, it must be generated with the 8-bit CPU option on !

2.4 *Use of 'Declarations' and 'Comments'*

An exemplification on the use of the '`Declaration`' symbols:



- The contents of unconnected '`Declaration`' symbols (a connected '`Comment`' does not count) are placed on the top level of the C-file and should be used for #include, #define and type declarations. Variables will be global.
  Best use only one of such a declaration (though multiple symbols are possible) for the order in the source code is hard to predict.
- Declarations may be connected to a '`Start`' symbol. In this case the content of the symbol is placed inside the function body of the process or procedure. Use this only for variable declarations.
- ☞ '`Declaration`' symbols with a connection to a '`Comment`' symbol is treated as 'unconnected'

Example: