Hi Phreaks.

This is a somewhat reduced copy of the original website dealing with the A1018s Handy ...

# http://www.spletomat.com/tech/s\_files/A1018s

Hope you find it as interesting as I do.

Ullasmann

The A1018s' architecture is almost identical to GA628. The ports are the same, only shifted up by 0x0800. The obvious advantage - a graphic LCD - greatly expands the phone's usage potential.

## Phone specs:

- AVR RISC processor @ 13MHz
- 1Mb flash (512x16)
- 32kb on chip RAM (no RAM chips on PCB)
- 8kb EEPROM
- 33x101 pixels graphic LCD

#### How to bootstrap the phone:

- power up the phone with +5V applied to test input
- phone will respond with "U" at 9600 bps
- send "OB" within 1 second (else phone turns on)
- phone will respond with "R"
- send two bytes 00 03
- phone will respond with two bytes 00 02
- send 64 bytes 00
- phone will respond with three bytes
- send another 64 bytes 00
- phone will send "S" after about half a second
- send the binary file to the phone
- Uploader will do all this for you

To see inside the phone, visit <u>www.inside-gsm.com</u>.

Please remember that everything here may not be 100% true and a lot is unknown. There is no datasheet for the phone - everything was researched by experimenting.

# 1. CPU

The CPU is a custom ATMEL AVR. MUL instructions are unsupported : ( and only basic LPM: r0 = (Z) and ELPM: r0 = (RAMPZ;Z) are implemented. Stack is post decremental - it points to the empty location below the last pushed value. The PC is 3 bytes wide. If you want to see where you are:

```
rcall _nexti
_nexti:
pop PChigh
pop PCmiddle
pop PClow
```

Onboard WatchDog is not present making the WDR instruction useless.

There's a couple of anomalies in the CPU. Instead of EIMSK and EICR registers there are RAMPX and RAMPY in their place. You can therefore address any memory location using X and Y index registers as well. Furthermore, the RAMPZ register is a bit special. When you change it, the load direct and store direct (lds, sts) instructions are also affected. XDIV register is actually EIND - the CPU supports EICALL.

# 2. Memory organization

Data memory:

RAMPZ (X,Y)	Range	What		
	0000-005F	AVR registers and I/O ports		
	0060-00FF	nothing. No internal RAM!		
	0100-01FF	hardware I/O ports		
00	0200-02FF  0F00-0FFF	14 mirrors of 0100-01FF		
	1000-2FFF	8kb RAM		
	3000-4FFF 5000-6FFF 7000-7FFF	.5 mirrors of 1000-2FFF (only half at 7000-7FFF)		
	8000-FFFF	32kb RAM		
01,02,03	0000-1FFF  6000-7FFF	4 mirrors of 8kb RAM		
	8000-FFFF	mirror of 32kb RAM		
047F	0000-3FFF	16kb RAM, mirrored on all above addresses and through to RAMPZ 7F		
808F	0000-FFFF	flash mapped to data memory (16 x 64kb = 1Mb)		
909F A0AF B0BF	0000-FFFF	3 mirrors of 808F		
COFF	0000-FFFF	a single byte mapped to the entire address space. It's writeable, but I dunno whether it's simple RAM or perhaps a register of some peripheral.		

## Program memory:

The available RAM is also mapped in the program memory - otherwise bootstraping wouldn't be possible. Bootstraping code to 4000 and executing the above code example produced program memory address 204000. The AVR uses the Harvard architecture concept - program memory and data memory are separated. Furthermore, program memory uses word addressing, whereas data memory uses standard byte addressing. For me, it took some getting used to and to make it simpler I wrote the following macro:

```
.macro ldz
ldi ZH,high(2*@0)
ldi ZL,low(2*@0)
.endmacro
```

It is used to load the Z index register with the right value when you want to point to some constant data you've defined in the program memory. Consider this example:

.cseg		
.org 0		
strDatal:	.db	"somedata"
strData2:	.db	"moredata"

Because of the word addressing, value of strData2 is 4 instead of expected 8. That's why I couldn't define strData1 as "some data" (with a space), since that would make strData2 4.5

ldz strData2 lpm

Loading Z correctly is made simple by executing the macro. The following lpm instruction then correctly loads "m" into r0. Identical macros ldx and ldy are defined as well (macros.inc).

## 3. LCD

Drawing on the LCD is ultra simple. It's identifier on the i2c bus is 0x70, after follows the line identifier, the x offset and finally 101 bytes of display data. Each byte defines 8 vertical pixels where LSB is the top and MSB the bottom pixel. The following transfer:

S 70 09 00 01 01 01 01 ...... 01 01 01 P would draw a line on top of the LCD. The identifiers for lines 1,2,3,4,5 are 0x09,0x29,0x49,0x69 and 0x89. There is also another type of LCD used in the phone - for details on how it works, examine the low level procedures.

You can try drawing with LCD tester.

#### 4. Other hardware

All other hardware is almost identical to GA628 so I won't describe it again.

This is a quick tutorial to help you start coding asap. I expect you've read the <u>CPU and Memory</u> <u>organization</u> already - the minimum what you need to know is there. There are also four other things you need: <u>ATMEL's AVR studio</u>, <u>low level procedures</u>, <u>the skeleton code</u> and the <u>flasher</u>. Here we go! =)

#### 1. Skeleton code

jmp

The skeleton code you downloaded looks like this. Fix my h:\a1018\... path to your own directory preference.

.include "h:\a1018\lowlvl\macros.inc"
.cseg
.org 0x0

.include "h:\a1018\lowlvl\ints.asm" .include "h:\a1018\lowlvl\lowlvl.asm"

reset

```
.db "MAIN PROGRAM START"
```

#### reset:

	clr	r0		
	out	RAMPZ,r0		
	out	RAMPX,r0		
	out	RAMPY,r0		
	ldi	r16, <mark>0xff</mark>		
	out	SPH,r16		
	out	SPL,r16	;	SP = 0xffff
	rcall	phInit		
	rcall	intInit		
	sei		;	auto watchdog reset
main:				
	; Your	code goes here		
	rjmp	PC	;	endless loop

The following examples are code inserted in this skeleton outline.

#### 2. Keyboard example

First, we call keyGetKey which returns a byte identifier of the currently pressed key in r16. Next, the keyDecode function translates this identifier into a more intuitive character representing the key (1,2,3,4,5,6,7,8,9,0,Y,N,C,L,R,\* and #) and returns r16. Finally we simply compare r16 to an immediate value and branch somewhere if they match.

main:

rcall rcall cpi breq	keyGetKey keyDecode r16,'Y' _yespressed			
rjmp	PC	;	endless loop	

You can omit the keyDecode procedure and compare the result of keyGetKey with the (cryptic) key identifier. It happens to be 24 for Y. You can examine keyDecodeStr to get the rest of them. An important thing to remember here is that when no key was pressed, keyDecode returns 0 whereas plain keyGetKey returns 25.

# 3. Lights example

The lights example turns on the top green LED by setting the carry flag and calling ItGreen. The carry flag is a parameter for light procedures - if set, the corresponding light is turned on, if cleared, the light is turned off. Secondly we turn on the keyboard-LCD backlight and set its intensity to half.

main:

```
sec ; sec = set carry flag
rcall ltGreen ; clc = clear carry flag
sec
rcall ltBacklight
ldi r16,0x80
rcall ltSetIntensity
```

rjmp PC

# 4. Sound example

```
main:
    ldi r16,1 ; lowest volume
    rcall bpSetVolume
    ldi r16,0x55 ; a nice annoying frequency
    rcall bpPlayTone
    rjmp PC ; endless loop
```

To turn off the "noise", either set volume or frequency to 0.

# 5. Serial communication example

### 5.1 Send example

Executing the following code would produce "Z=5A" on the receiving end.

main:

ldi rcall ldi rcall ldi rcall	<pre>r16,'Z' comOutChar r16,'=' comOutChar r16,'Z' comOutHex</pre>		
rjmp	PC	;	endless loop

Note that comOut procedures first wait for the previous byte (if any) to be transmitted.

# 5.2 Get example

main:

rcall cpi breq	comGetChar r16,' <mark>X</mark> ' _xreceived	; blocking
rjmp	PC	; endless loop

comGet procedures are blocking - they return when byte(s) were actually received. If you only need to check the status of the in buffer, examine comGetChar to see how it's done.

# 5.3 Setting baudrate

main:

ldi rcall	r16, <mark>96</mark> comSetBaud	;	9600 baud	
rjmp	PC	;	endless loog	р

comSetBaud does very little work - it translates the "more intuitive" number to the hardware understood value and sets baud port. Valid values for r16 are: 48 for 4800 96 for 9600

19 for 19200 38 for 38400 57 for 57600 115 for 115200

### 6. LCD example

I reserved two memory buffers for LCD procedures. One is called lcdTextRam, the other lcdGraphicRam. They're both defined in lowram.asm (low level procedures' RAM). lcdTextRam is 68 bytes long and if you put some ASCII characters in it and call lcdUpdText, these characters will appear on the LCD. On the other hand, lcdGraphicRam is 505 bytes long, calling lcdUpdGraphic will draw the pixels defined within it.

## 6.1 LCD text example

The following example copies a constant string from program memory (flash) to lcdTextRam using a helper procedure and then draws the text. The helper procedure - lcdCopyTextStr - takes 3 parameters, a pointer to the string in Z, the length of the string in r16 and the offset in lcdTextRam to copy to in r17. Since the string is 68 characters long, the only valid offset is 0. If you used a shorter string - strShorter - and put 17 to r17, ERICSSON would appear in the second line.

main:							
	ldz	strHello					
	ldi	r16, <mark>68</mark>	;	length of	string		
	ldi	r17, <mark>0</mark>	;	offset to	copy to		
	rcall	lcdCopyTextStr	;	copy text	string f	rom pro	gram
	rcall	lcdUpdText	;	memory to	lcdTextR	am	
	rjmp	PC	;	endless lo	qoo		
line strHello:	1	line 2	lin	e 3	line 4	I	
.db	"HELLO WOR	RLD					н
strShorter	<u>-</u> :						
.db	"ERICSSON'	1					

#### 6.2 LCD graphic example - PutPixel

This example expects lcdGraphicRam to be uninitialized, so it calls lcdClrGraphic first. Next, we put a single pixel in the top-leftmost corner of the LCD and call lcdUpdGraphic to update LCD with this change in lcdGraphicRam.

main:

rcall ldi ldi sec rcall rcall	<pre>lcdClrGraphic r16,0 r17,0 lcdPutPixel lcdUpdGraphic</pre>	<pre>; x coordinate (0100) ; y coordinate (032) ; put pixel, clc would clear it</pre>
rimp	PC	; endless loop

#### 6.3 LCD graphic example - constant picture data

You can also define an entire screen of pixels in flash, copy them to lcdGraphicRam using lcdCopyGraphic and draw it. There's a simple tool I wrote called <u>BMPconverter</u> that will produce an array of values like you see below, from a 33x101 1 bit bitmap.

main:

ldz	graEricLogo
rcall	lcdCopyGraphic

rcall lcdUpdGraphic

rjmp PC ; endless loop

graEricLogo:

```
.dw 0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000
.dw 0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000
.dw 0x0000,0x8000,0x8080,0x0000,0x0000,0x0000,0x0000
.dw 0x0000,0x0000,0xF0F0,0x30F0,0x3030,0x0030,0xF0F0,0x30F0,0xF030,0xF0F0,0xF000
.dw 0xF0F0,0x8000,0xE0C0,0x70F0,0x3030,0x3030,0xE000,0xF0F0,0x3030,0x3030,0xE000
.dw 0xF0F0,0x3030,0x3030,0xC000,0xF0E0,0x3070,0x7030,0xE0F0,0x00C0,0xF0F0,0xF0F0
.dw 0x80E0,0xF000,0xF0F0,0x0000,0x0000,0x0000,0x0000,0x1800,0x3C3C,0x9E3C,0x9E9E
.dw 0xCFCF, 0xE7CF, 0xE7E7, 0x00C3, 0x0000, 0x0000, 0x0000
.dw 0x0000,0x0000,0x3F3F,0x333F,0x3333,0x0033,0x3F3F,0x033F,0x1F03,0x383F,0x3F00
.dw 0x3F3F,0x0700,0x1F0F,0x383C,0x3030,0x3030,0x3000,0x3331,0x3F33,0x1E3F,0x3000
.dw 0x3331,0x3F33,0x1E3F,0x0F00,0x3F1F,0x3038,0x3830,0x1F3F,0x000F,0x3F3F,0x003F
.dw 0x0F03,0x3F3E,0x3F3F,0x0000,0x0000,0x0000,0x0000,0x6000,0xCFCF,0xE7CF,0xE7E7
.dw 0xF3F3,0x79F3,0x7979,0x0030,0x0000,0x0000,0x0000
.dw 0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000
.dw 0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000
.dw 0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000
. dw \ 0x0000, 0x00000, 0x0000, 0x000, 0x0
. dw \ 0x0000, 0x00000, 0x0000, 0x000, 0x0
.dw 0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000
```

Note that there's an extra 0 at the end of each 101 byte "line". That's because of the word addressing - you can't define an odd number of bytes.

### 7. EEPROM example

Both EEPROM routines eeRead and eeWrite use register pair X as EEPROM address parameter. Since EEPROM is 8k bytes, values 0..1FFF are valid.

# 7.1 EEPROM reading

```
numEntries
                        =$10FF
.equ
main:
        ldi
                XH, high (numEntries)
                XL, low(numEntries)
        1di
                eeRead
        rcall
        cpi
                r16,30
        brlo
                _storeentry
                                ; less than 30 entries
        rjmp
                PC
                                 ; endless loop
```

The example demonstrates how you could define an address for a value in EEPROM and retrieve it using eeRead.

# 7.2 EEPROM writing

main:

ldi ldi ldi	XH,0x10 XL,0xff r16,30		
rcall	eeWrite		
rjmp	PC	;	endless loop

Here we overwrite our hypothetical value numEntries from the previous example with 30 using eeWrite.

# 8. SIM card communication

Like the LCD, there's a buffer defined in lowram.asm for SIM comm called simBuf. It's 255 bytes long and gets filled by info from the SIM card when you call simRead. The number of bytes in the buffer is stored in a memory variable called simBufSize located just after the buffer. I expect you're familliar with SIM card commands and file organization.

## 8.1 Resetting the card

SIM cards come in two flavours: inverted ISO and straight ISO. Before doing a reset, specify which card is in the slot by calling either simInvISO or simStrISO.

main:

```
rcall simInvISO ; we have an inverted ISO card
rcall simATR
lds r16,simBufSize
rcall comOutHex
rjmp PC ; endless loop
```

simATR powers up the card, resets it and calls simRead immediately after, in effect filling simBuf with card's ATR message. The example above resets the card and sends the number of bytes in card's ATR message across the serial interface.

## 8.2 SIM select file example

Note that this example expects you've already selected card type and reset the card (i.e. what we did before).

main:

ldi	r16, <mark>0xa0</mark>	;	CLA - GSM SIM card
rcall	simSend		
ldi	r16, <mark>0xa4</mark>	;	INS - SELECT FILE
rcall	simSend		
ldi	r16, <mark>0x00</mark>	;	P1
rcall	simSend		
ldi	r16, <mark>0x00</mark>	;	P2
rcall	simSend		
ldi	r16, <mark>0x02</mark>	;	Р3
rcall	simSend		
rcall	simRead		
rjmp	PC	;	endless loop
	ldi rcall ldi rcall ldi rcall ldi rcall ldi rcall rcall rcall	<pre>ldi r16,0xa0 rcall simSend ldi r16,0xa4 rcall simSend ldi r16,0x00 rcall simSend ldi r16,0x00 rcall simSend ldi r16,0x02 rcall simSend rcall simRead rjmp PC</pre>	<pre>ldi r16,0xa0 ; rcall simSend ldi r16,0xa4 ; rcall simSend ldi r16,0x00 ; rcall simSend ldi r16,0x00 ; rcall simSend ldi r16,0x02 ; rcall simSend rcall simRead rjmp PC ;</pre>

The example shows the beginning of a select file command. After simRead, if everything was ok, simBufSize should equal 1 and value A4 should be in simBuf as the card acknowledged select file command.

This concludes the tutorial. I haven't covered all procedures in lowlvl.asm - you can find their "prototypes" at the beginning of the file. I believe they are straightforward. You can always find more example code on this site - like <u>Tiny Arkanoid game</u>. If there's something you need help with, ask in the <u>developer forum</u>. And finally, I would love to hear about anything interesting you code for the phone, so be sure to let me know. Good luck!

The chatboard is a silly little thing that virtually presses keys on A1018s using AT commands.

Here's how it works:

- First the chatboard keeps sending **AT** until it receives OK from the phone.
- Secondly it sends **AT+CGMM** and **AT+CGMR** requesting model and revision information
- Finally keys are pressed utilizing the **AT+CKPD="X"** command where X is one of the following:

X	key
09	numeric keys
# * < >	hash, asterisk, left and right
S	YES
e	NO
С	CLR

Characters are obtained simply by pressing the corresponding numeric key required number of times. Several keys can be pressed with one command like this: **AT+CKPD="12345"** 

The chatboard makes use of the phone's logic level reference pin making it hardware compatible with GA628, however the phone's firmware doesn't support it.

I wrote an emulator for the chatboard for those of you who would like to write SMS messages quickly.

Thanks to Ciril for dismembering his chatboard and letting me have it for research. =)