Several efforts were made from chip manufacturers (e.g. Zilog, Z8) till third party companies (e.g. Parallax, Basic Stamp) to constitute an easy-to-use device for non-professional users.

Our attempt to achieve this goal is **A**nother **B**asic **S**ystem, to be abbreviated and mentioned later as ABS.

Main disadvantage of similar devices is the absence of correct timing possibilities. Using loops for timing purposes is neither exact, nor handy. ABS is free from this disadvantages due to an unique realtime background. The easy-to-use end result is built upon three level time handling features to be discussed below.

Considering expected features of a controller for hobbists are:

- *simple program language*
- *standardized data length*
- *expandability*
- *interactive testing during development*
- *no hardware for programming*

Among conventional languages FORTH is very near to these requirements: standardized data and expandability are excellent attributes. Unfortunately RPN (Reverse Polish Notation) approach makes it rather uncomfortable for hobbists. To overcome this bottleneck we wedded FORTH with BASIC to get a BASIC-look environment and straight arithmetic notation. The resulted ABS is in the *internal* structure a stack-based FORTH machine, but the user sees it as a BASIC environment. This way user is saved from using RPN (Reverse Polish Notation) arithmetic. Unified 16 bits wide word representation of all variables, constants and operators eliminates the need for type conversions, resulting easy use and fast operation while all „manual" stack-based statements were eliminated. Nevertheless FORTH's advantages were kept inside as follows:

- fixed length data eliminates the need for data type matching
- ability of secondary declaration was kept

Moreover secondary words are compiled as a list of machine-code calls, avoiding circuitous mechanism of FAST secondary word pickup. Nevertheless the secondary building syntax is hidden behind the usual SUB (subroutine) known from BASIC language. There are formal syntax equvivalences between ABS and FORTH. A secondary is declared in FORTH looks like this:

: *myfunction* statements **;**

(note: For those not familiar with FORTH secondary declaration, is suggested to think of it as a named subroutine. As you declare it will become immediately part of the language and has the same scope and rights then built-in commands.)

For an ABS declaration you will use the same structure, substituting starting '***:***' with '***sub***' and closing '***;***' with '***endsub***' this way;

sub *myfunction* statements **endsub**

The subroutine is called simply by its name. Entering *myfunction* will run all the statements enumerated between *myfunction* name and closig keyword **endsub**

Suppose you have a red LED connected to bit0 of portd. Then you declare a command as follows:

sub red portd=1 endsub

Here sub introduces a new command to be named *red*. The body of command is rather simple, just sets bit0 of portd. At last *endsub* keyword closes declaration.
When it is done, it is enough to execute *red* and red led will be switched on. That's all...

16 bits wide word representation is adequate for control as can be seen at FORTH, but there is a point where it appears narrow bottleneck. At calculating some rate the product of a multiplication may be truncated and subsequent division causes even a more catastrophic result. To eliminate this distortion FORTH introduced */* operator. ABS uses a flawless solution to avoid this problem without using any special operator. Upper part of the multiplication product (i. e. bit31..bit17) is put into a hidden overflow variable. Subsequent division is then performed on dividend composed of original dividend completed with this hidden overflow variable. No special effort needed to clean overflow variable when the product fits in 16 bits as the zeroed bit31..bit17 cleans it.

There are some internal variables are available for users. These variables can be written and read as other variables, but they influence the operating system running in background. Some of them may even be modified by the system. These are:

timer0
timer1
timer2
timer3

As names suggest these user variables act as timers. Each of them count backward. At bootup stepping rate is set to 1 Hz, i.e. each of them steps down at each second. Reaching 0, a turn to 0xffff (or 65535 if you like it better) will take place and counting goes on.
Timers can be set to expected delay time. Wring e.g. 10 to timer0 (simply write timer0=10) your timer0 will count back 10 seconds. During this time you may do anything else provided sometimes check if timer0 equals to zero. Due to the system quartz crystal referenced timing you get three advantages:

- you are free from running useless delay loops for timing
- you can handle unexpected external pin events during delay
- however your timer0 polling may jitter in time, the timer runs without any jitter, resulting stability of a quartz controlled clock – because actually it ***is*** *a quartz controlled clock.*

Each timer is independent and have no influence to others. More about system variables belonging to timing system will be discussed later.

Now you know enough to write your first application, a traffic lamp control. Connect a

- *red led to bit0 of portd*
- *yellow led to bit1 of portd*
- *green led to bit2 of portd*

Now declare your commands. You will use them later to build up your application.
First of all, enable portd to act as output by setting data direction register (ddrd) of portd.

ddrd=0xff

Remember you have a timer decreasing content from second to second. To catch moment when timer changes declare a wait command as follows:

sub wait a=timer0 : while (timer0 == a) {} endsub

You save the momentary value of timer0 into variable ***a*** and are continuously checking whether timer0 is still equal to saved value stored in ***a***. While these are equal, continue checking. When there was a step down in timer0, your loop ends.

Now build a command for 2 seconds delay.

sub delay2 x=2:while (x) { wait:x=x-1 } endsub

You put 2 into ***x*** variable and... use your *wait* command. What is great, you may use your just defined command encapsulated into a newer program. Running *wait* you decrement ***x*** variable as well, so *delay2* will be finished after calling *wait* twice.

There is nothing new in declaring a 5 seconds delay.

sub delay5 x=5:while (x) { wait:x=x-1 } endsub

You remember how red lew was switched:

s*ub red portd=1 endsub*

Based upon your experience with red led, yellow led on bit1 of portd is straightforward:

sub yellow portd=2 endsub

You will need a phase when both red and yellow leds are activated:

sub redyellow portd=3 endsub

To switch green led on bit2 of portd is not a challenge any more:

sub green portd=4 endsub

Now you have commands built *exactly for yor needs*, the rest is still easier:

```
while (1)
{
red : delay5 : redyellow : delay2 : green : delay5 : yellow : delay2
}
```

That's all. Easy, isn't it?

Of course this is not a complete application, just a phase of traffic control. For real project you have to care about cross direction lights, build in sensor acception, exraordinary case handling, etc, This short example was cited just to introduce the power of language

expandabilty.

As you see, complex sequences can be packed into single command and these just defined commands may be packed into other command in endless depth. This way your commands became more and more compact.

Timing is of primary importance. ABS supports timing tasks in different levels. First level of timing possibilities is based upon the use of timer0...timer3 variables. These seems to be simple variables for the user, they can be written and read at any time. However they are handled by the operating system as well as counters. Each of them count back in a rate prescribed by scale0...scale3 prescalers. These are also user variables and also can be written and read at any time. scale0 belongs to timer0, scale1 belongs to timer1, etc, respectively. Any value stored in scalex can be thought as the number of milliseconds (1/1000 part of a second) units. E.g. writing 1000 into scale0 will cause timer0 decreased at each second. It will happen independently of main program, no matter how busy the user program is.

This way the user is at leisure and he/she may check unhurried the timers. Timers keep valid time and this time is as exact as system quartz crystal is. The only price to be payed for it just to poll registers a bit more often then time elapses. When expected time is reached, task can be done. It can be adequate in several cases, e.g. running a digital alarm clock written in fully BASIC.

Next level of timing possibilities makes timing more comfortable and reduces timing jitter into the order of a few ten nanoseconds.

In contrast of level one, there is no need for polling. A BASIC subroutine can be assigned to any of timer0...timer3. Any time the timer steps, subroutine is called and executed by the system. During this call the execution of running program suspended as long as the subroutine is running. It also limits the lengh of subroutine execution time. Due to internal system parameters, it cannot be longer than 0.1 msec. However it is enough to execute hundreds of calculations and port handling. Care should be taken to eliminate time consuming operations, e.g. debounced key readings.

The most severe timing is absolutely exact. A special pin can be set/reset at pre-programmed time. It is operated by dedicated hardware part of the controller and it has priority over all operation. If the preset time is reached, the pin will take prescribed change even if the controller is being executed an interrupt is or even in the middle of a 2 cycles long execution. Upon this feature e.g. a frequency counter gating can be performed, resulting unique possibility to build an intelligent frequency counter – written in BASIC.

Debugging

The potentiality to build up your program step by step by defining your own instructions from simple level to more complicated makes debugging easier. However you have a need to examine or overwrite variables during debug process.

Warning! *Care should be taken to distinguish* ***commands*** *from* ***statements****.*

When you write your program, you write ***statements*** to be downloaded, stored and executed later by the chip. On the other hand when you ask for the content of a variable, that is a direct ***command*** and will not be stored. So be careful using it.

Read and write commands have different syntax. To examine the content of a variable or a port, simply write the name of it followed by a question mark.

Note: *this form always represent a command, will not be stored, but will be executed*

immediately. E.g.

myvariable ?

The content will be reported in decimal, hexadecimal and binary form as well. Ports also can be invastigated this way. E.g.

ddrd ?

will report how data direction register of 'd' port is set.

It is possible to set variables or ports by command, but first clarify the two ways of writing variables or ports.

The write ***command*** for variables or ports are preceeded by the ***!!*** sequence. Double exclamation mark is used to distinguish commands from statements. To write for debug purposes a variabe or port immediately you should write

!! myvariable = 112
or
!! pord = 7

Remember, these are ***commands***, i.e. *immediate events* and will be executed as you issue them. The same pattern without leading ***!!*** sequence will be interpreted as statement, will not be executed immediately but will be stored instead as part of your program.

One more debug utility is to execute any of the statements you have defined by the *sub … endsub* sequence. To execute it as a ***command***, use the syntax you already used at variable write. So if you have defined earlier *mydemo* as.:

sub mydemo ddrd=0xff : ddrb=0xff : portd=0x55 : portb=0xaa endsub

then you have the possibility to run

!! mydemo

to enable B and D ports as outputs and write prescribed values to the ports. Remember this example and remember you have *possibility* to run – but before you try it you have to ***save*** the declararion. *As long as it is not saved, the declaration exists just in your computer, your ABS chip dos not know anything about it*.

This is an often comitted mistake. Any time you find something is wrong, check yourself whether was code downloaded or not. ABS chip cannot guess statements existing just in your computer.