The Concept

This is a mix of C, BASIC and others language that sits on the microcontroller and offers the following advantages that alter approach to development.

* The language is initially interpreted but any created functions are compiled thus you get all the advantages of an interpreter such as instant results being able to examine actual registers and memory addresses and being able to run individual functions. At the same time when a function is created this is compiled and so is extremely fast as compared to a standard interpreter.

* Because the functions are compiled and there is direct access to the registers there no need to provide hardware type functions such as SPI or I2C as these can easily be created by the user if and when required. There is now a suit of functions called rookie (see library) that does just this.

* All functions are created as text either on the IC or downloaded as a text file. This gives the user, any user, not just those with programming tools, the power to change and modify existing code. The code is easy to read and sometimes all it takes is a minor modification to make all the difference to a hardware project.

* Because everything is available it is very easy to carry out 'what if' scenarios. All of the code is stored in RAM until the user decides to move it to Flash to be more permanent, even then it can be erased later.

* As code is run on the actual IC itself this makes an ideal learning platform, setting and reading registers, connecting other hardware to see the effects. The syntax uses C like operators and so is a good starting point for learning other languages. It can be difficult to understand what and how the registers effect the hardware modules from reading the data sheet. Being able to change them and see the effects immediately greatly enhances this procedure.

* Because the functions are compiled they are relatively fast and so useful for signal capture and the like. Not as fast as a compiled language but much faster than normally interpreted languages. The speed is approximately 500,000 lines per second. As the microcontroller contains useful hardware modules often it is only necessary to provide the control so speed is not such a great issue. PWM is a classic example where, once the module is set up, only a register needs setting to control the output.

ByPic is an interactive compiled language designed for use with a microcontroller. It sits in the microcontrollers Flash memory and can be extended by user input. It is a threaded

Search

Search

What Is It
Language Guide 2.30
Key Words
IDE New
USB to Serial
ESP8266
FAQ
Error Codes

External Links

Creative Science Centre

language which takes text as input and then compiles this to the processor memory in the form of addresses.

Why Use it

It can do anything C or other languages can do. The main advantage is that it is interactive which means that you can inspect the microcontroller registers in real time. It is easy to use and learn but most of all it is easy to experiment with. A big advantage is the speed of development which is much faster than the code, compile, download method used in the traditional systems. All development is done in RAM and if a particular function or set of functions is used many times then it can, at any time, be saved to Flash. Only when the function is fully developed and tested does it need to be saved to Flash - this is in complete contrast to the code, compile, download method of working.

Another major advantage is that if a hardware constructional project is presented, the code can be changed if required by the final user because all that is required is text.

The basis of the language is functions. These are built in and can also be created by the user. A function to say read a GPS, once written and debugged can be saved to flash and then used in an application. An application simply consists of many functions called by a main function – exactly the same as C.

The main advantage here though, is that any function can be called at any time (interactively) so the GPS function for example can be called form the terminal to read the data and check if it is working or returning the expected data. This makes programming much easier and more fun.

What Can It Do

Microcontrollers are configured and operated through their registers. ByPic allows direct access to these. For example to examine the contents of a register at memory address 0xbf886040 simply use peek or if it is defined:

constant PORTA 0xbf886040

print @PORTA

The above will print out the register contents. Similarly setting interrupts times and all of the other peripherals uses a similar technique. This gives the same control over registers as you have in C or assembler but with the immediacy of being able to manipulate, set and see them.

In addition to all of this there are floating point data types and other data types for holding co-ordinates for touch screens.

Speed

This is an not an interpreted language and its not a compiled language language in the conventional sense. To explain; a conventional compiler takes the code as text and then creates code that is downloaded into the microcontrollers Flash memory. This language will

take the source text and compiles it into addresses. When the program is running the addresses are executed one by one, these addresses point directly to machine code and thus the program runs as fast as the address pointer can point and run the addresses.

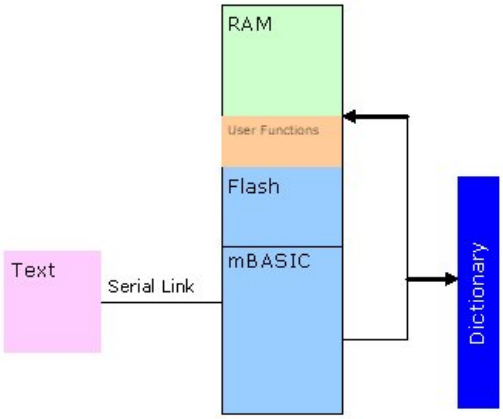
Benchmarking on an 80MHz processor shows that the program executes at in excess of 400,000 lines per second. It depends of course what the 'line' is doing but this equates approximately to 2.5uS per line.

Because the internals of the program 'runs addresses' if a further increase of speed is required a C or assembler function can be written and this can be run, this is called a C Plugin.

What Does It Run On

At he moment it is only compiled for running on the PIC32 and then it will only load via the boot loader provided with the ByVac PIC32 microcontrollers. There is no reason why this cannot run on other microcontrollers providing there is at least 256k Flash and 32k RAM. There is a cut down version (no SD Card interface) that requires approximately 100KB. As of August 2012 this is now available of the PIC32MX1 family that only has 128k Flash. This version is not boot loaded but permanently resided in theMX1 Flash. There is still room for about 2000 lines of user code.

How does it work

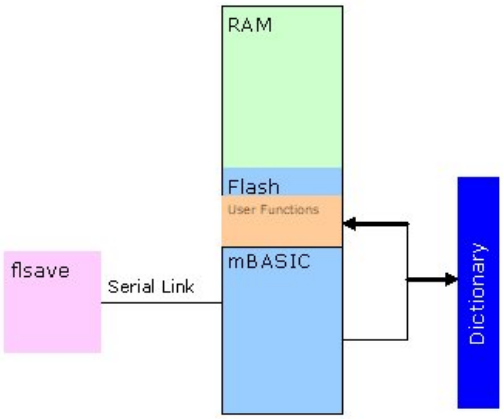


All ByPic devices have a serial link which is the main method of communication. The user can type directly into the device or pre-prepare functions in a program editor. When ByPic receives the function it will check the syntax and compile it into RAM as shown. There are many predefined functions and if any are encountered then the address of the encountered function is also compiled into the user function. A dictionary of functions is maintained by mB, if a function is not defined and it is used then an error will occur. As an example a predefined function is hex$() which converts a number into a string in hex format. This could be defined by the user but as it is used quite often it is one of the built in functions.

If the user as part of a user function uses hex$() then ByPic will search through the dictionary and when found will compile its address into the user functions. This scheme is self perpetuating as when a user defines a new function, that too can be used in further functions.

Saving

The main point of this scheme is to enable the user to built up a list of functions appertaining to the required project. In this way the project can be verified, experimented with and than ultimately released. A robot arm for example could have a function called wrist(degrees). This would be a use function and calling wrist(20) would turn the wrist 20 degrees. This can be typed in at the keyboard to verify everything was okay. A complete set of these functions could be saved and then higher level functions could use them; e.g pick_up(item).



To carry out the above ByPic can save the user functions to Flash and they simply become part of the language and can be used in the same way as the hex$() function example given above. This also leaves the RAM free to develop further functions. Another advantage is that as the application grows the loading time does not as all of the tried and tested functions are in Flash.

History

Introduction

This language has been developed on the experience and feedback from the original BV_Basic. The primary use is for microcontrollers that have limited resources and require hardware control. The language is built on the following principles:

- Interactive: the user should be able to interact with the connected hardware, trying out different things without the need for a lengthy compile and download.
- Tools: no special tools should be required so that it can be used with any host that supports a serial port.
- Extensible: the language supports user functions that allow the language itself to be extended to fit the user application. This is done by the use of functions that are written an can be saved to Flash.
- Fast: nearly all of the error checking is carried out at compile time. This means that running code only needs minimal error checking. The code can be saved as a list of addresses that enables the fastest possible interpreted language.

Background

This language came about because of my interest in Forth. For those who have never heard of this language it was invented in the 1960's by Charles Moore (http://en.wikipedia.org/wiki/Charles_H._Moore). Back then compilers were just beginning to be invented and nobody knew the best way to program computers. The concepts that Forth used were remarkable and in my opinion still are. It has a sort of elegance that no other language has, this is because there is a very close link to the way in which computers work and the way in which the language works.

When I first discovered this language (1970's) it looked full of promise. Migrating from assembler to Forth felt like going from a Mini to a Rolls Royce the power was truly amazing. There was a speed penalty but it was well worthwhile. It was a difficult language to learn but that was offset because you could achieve a considerable amount only knowing how a small portion of the language works.

The downfall, for me anyway, was the actual code; the text which forms the code. There was a joke that Forth is write only; its not a joke its true. Because Forth puts efficiency above everything else the programmer needs to do a lot of mental manipulation of code elements, in Forth's case the stack mainly. This leads to code that is easier to re-write than modify. Ways round this have been proposed such as writing the code in very small chunks. Anyway its not my intention to knock the most interesting language ever invented.

Having fallen out with Forth initially I kept going back to it time and time again in cycles of a few years. Initially thinking this is such a great language and then realising why I had left it before. So what's wrong? Well, when I can't understand something I wrote in Forth only 6 months ago there must be a better way? I have been thinking about this for some years and have made a few half hearted attempts at solving the problems associate with Forth but never got anywhere.

Here is a list of **Forth Language** good and bad points

Good points

- Interactive
- Comparatively fast
- Extensible

Bad Points

- Unreadable code
- Only one data type
- Difficult parameter passing
- All global – difficult to modularise

Interactivity has to be the main strength, to be able to see variables there and then and to be able to control some machine by typing a few words just can't be beaten, particularly in a learning environment and for me that's all of the time.

Forth is quite fast for an interactive language, not as fast as C or assembler but not that far behind, some simple tests show that it is about 7 times slower which is not bad particularly when hardware is getting faster all of the time.

Extensible is probably not the correct word to use, Forth is extensible in that it has defining words (*a word in forth is the unit of language like a function*) that enable other words to be written as if creating another language. This facility, although very cleaver is not much use, if it were, the modern languages would have it – its simply not needed but at the time who knew that? Even so the way that an application is put together is a kind of extensibility as the user words effectively become part of the language.

As mentioned before no matter what effort is made the code is unreadable. Even writing very small words doesn't help much as you still have to pass information from one word to the next and there is only that stack or global variables to do that. Two main contributors to unreadable code is the lack of local variables so you have to manipulate the stack and the second is not having a 'writeable' (so it can be read back) method of passing data into and out of words.

In The Beginning

I am sure that everybody who has come into contact with Forth for an extended time thinks about 'Fifth' its a language just begging to be improved, building upon such a great concept. The idea is to have something like Forth but without the cumbersome syntax. One idea is to write a new language using Forth; the Forth will interpret the new syntax and make Forth words from the new syntax words. For example

print "fred"

The above could be interpreted in forth and the output would be something like ." fred" type. On the face of it this looks quite promising and in fact C. Moore did actually produce a BASIC compiler using Forth. Another advantage is that the compiling could be done outside the target, but wait! This is simply a BASIC like compiler of which there are many. This does not have the interactivity of Forth and so this option is a none starter.

Having started with something like this idea and over many iterations what is the result? What is the difference between a standard tokenising interpreter an ByPic? On a tokenising interpreter the text file which is the BASIC or other language source code is broken down into tokens, the tokens represent elements of the language that are stored in an efficient manner, 'print' for example may be stored as a single byte. When the program is executed, an interpreter converts the tokens into meaningful instructions and then depending on what they are will eventually call a functions that actually does the work. If we convert this effort into time we have:

- Part 1 convert text to tokens (interpret time)
- Part 2 convert tokens into functions (run time)

The way ByPic works is nearly that all the effort is spent on Part 1 and instead of converting the BASIC code into tokens it converts the code into actual function addresses and so 'print "fred"' would be stored as the address of the actual print function followed by the text "fred". The effect of this is to remove the time taken in part 2 to interpret anything. It also means that at any time a function created can be run independently and so making testing and debugging easier.