Bare-Metal Benchmark Suite

BMBS

October 23, 2017, Version 1.0

Author: Alexander Wachter
Source Code: https://github.com/alexanderwachter/bmbs

Abstract. Benchmarks are used to evaluate the performance of a device. Today's benchmarks are focused on comparing high performance CPUs and relay on an underlying operating system. Most of them are not designed to run on microcontrollers. There is no benchmark suite for cross-compiling different benchmarks on different platforms without an operating system. BMBS is a suite for writing platform independent benchmarks, to cross-compile and link them together to a single binary and flash it to the target device. The user can combine his desired platform, one or more benchmarks and a toolchain. This makes BMBS suitable not only for checking CPU performance but also the performance of a specific algorithm or toolchain. There is also a host-tool to communicate with the device connected. With this host-tool it is possible to reset the device, list the flashed benchmarks, start them independently, collect the results and show print statements sent with a print command from the running benchmark. Thanks to a well defined command structure this host-tool is platform independent and does not have to be recompiled for different platforms. For evaluation a hardware abstraction for a 8 bit AVR (ATmega328p), a 16 bit MSP430 (MSP430F5529) and a 32 bit cortexM4 (LM4F120) is already implemented.

Table of Contents

1 Introduction

Benchmarks are used to measure the performance of a CPU in terms of how fast a specific piece of code can be executed. This is useful to compare different CPUs of the same family or different CPU families with each other.
Some benchmarks make heavy use of floating point arithmetic, some use fixed-point arithmetic and others are very complex in terms of control flow. It's important to know what the benchmark is actually measuring and what one wants to measure.
Benchmarks help deciding what kind of CPU or even which specific CPU is most suitable for a given task.
Most of the benchmarks existing today are very resource consuming in terms of memory and rely on operating system support. This makes them hard to use on microcontrollers which lack an operating system and only have a few kilobytes of data and program memory. Whetstone [3] for example is quite small and does not use file IO or other operating system calls and therefore could be used on microcontrollers but it mainly tests floating point operations and nothing else. Another famous benchmark is Dhrystone [10]. It can be seen as an integer alternative to Whetstone but it uses a 50 times 50 integer matrix which exhausts most small microcontrollers. A newer benchmark is CoreMark [4]. It is designed to be an embedded benchmark which implements various use cases like matrix multiplications or list operations. CoreMark is quite simple to port on different platforms by just implementing some predefined platform depending functions. This hardware abstraction is nevertheless very close to the benchmark code and includes all separated benchmarks within the main function. This complicates writing and testing specialized benchmarks. The lack of an open source license restricts the use of CoreMark in education or research.
All benchmarks existing today consist of a fixed set of tasks which can not be compiled and run separately. Moreover they are designed to compare hight performance CPUs with lot of resources in a very generic way. Embedded systems with specialised CPUs are very performance critical. Therefore, attention must be paid to select one that fits the needs but is not oversized. An oversized CPU can cause problems with battery lifetime and cost. The generic benchmarks existing are not suitable to compare microcontrollers with dedicated algorithms to find out which one fits this special needs best.

In this work a bare-metal benchmark suite is created - a special suite designed for running benchmarks on microcontrollers without an operating system. The Bare-Metal Benchmark Suite, short BMBS, can compile and run new or existing benchmarks by providing a hardware abstraction layer, a host tool and useful library functions which make the benchmark code independent of hardware and toolchains.
After implementing a platform dependent layer it is possible to run all provided or self written benchmarks, as long as there is sufficient memory on the target device. The platform only needs to provide at least one hardware timer and one

UART with interrupt support. With the host tool it is possible to start selected benchmarks, to collect results, reset the device and show print statements from the device under test.
The provided build environment, based on CMake [9], is able to combine a list of benchmarks with a chosen toolchain as well as a chosen platform. Therefore it is possible to have various combinations of benchmarks, platforms and toolchains to test. BMBS provides the opportunity to select the single benchmarks as well as the target-platform which should be included in the build. The order in which the separate benchmarks are executed can be chosen from the host during runtime. Because benchmarks use a registration macro within their own C file it is neither necessary to make any changes in the existing code nor in the abstraction layer when writing a new benchmark. Thanks to its design BMBS is therefore suitable for testing specific algorithms on various platforms. This is useful to help selecting a platform for a new design of a embedded system in terms of execution time of a dedicated algorithm. Furthermore time independence of algorithms, which is an important property of cryptographic algorithms, can be proven.
The host tool is used to control the device connected. It can discover the benchmarks currently programmed, start one, a list or all of them, collect the results and reset the device. Because the connection is based on a special protocol, which is implemented in the platform independent part of the software running on the device, the tool is independent of the platform connected. For this reason it is not necessary to rebuild the host tool for different platforms. For evaluation purpose the following platforms are implemented:

- ATmega328p A widely used 8 bit RISC microcontroller from ATMEL
- MSP430F5529 A low power 16 bit microcontroller from Texas Instruments
- LM4F120 A 32 bit cortex M4 microcontroller from Texas Instruments

The remainder of this work is structured as follows. Section 2 is a high level overview about the systems components. Section 3.1 brings a deeper overview about the system architecture. The build process is annotated in section 4, the host-device communication and the protocol are annotated in section 5 and the last section 6 contains the evaluation results.

2 High Level Overview

A general setup for running benchmarks on a microcontroller consists of following:

- A host PC for running the host tool, building the binary and programming the device.
- The device under test running the benchmarks.
- A serial connection between the device under test and the host PC.
- A programmer for flashing the program to the device.

This is illustrated in Figure 1. When a desired platform is evaluated the first thing to do is running a CMake build. With this build the desired target platform, the toolchain and the single benchmarks are selected. After this a cross compilation is started by running GNU MAKE [5]. This puts all the necessary library functions, platform abstractions and the desired benchmarks together. The result is a binary file ready for flashing to the microcontroller. By running a "*make download*" command the previously built binary is automatically programmed to the connected device. The device is now fully prepared and ready for connecting. To run the benchmarks and to collect the results the host tool must be started with the COM port to which the device is connected to as an argument. When connected the host tool shows a prompt where the commands for showing the programmed benchmarks, running them, printing the results and reseting the device can be typed in.

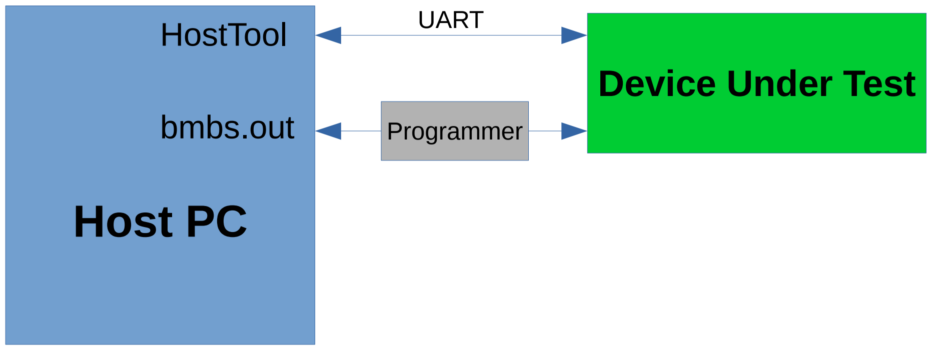


Fig. 1. General Setup

3 Architecture overview

The projects structure is designed in a way that one implementing a benchmark doesn't have to care about the underlying platform. This is done by abstracting away the platform dependend code from the benchmarks. When a new benchmark is implemented only functions provided by the library should be used to guarantee platform independence. An overview of the directory structure is given in Figure 2.

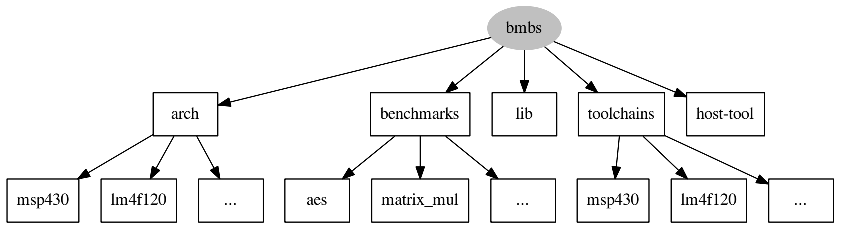


Fig. 2. Architecture overview

3.1 Arch

The abstraction layer arch provides the implementations for platform dependent functions. For every platform there must exist an implementation for every function prototype of all header files in the include directory from the arch directory. The code of this implementation must be placed in one single subdirectory of the arch directory. This is because the toolchain only compiles the subdirectory dedicated to the selected architecture. The naming of this subdirectory defines the name for this platform. The choice which subdirectory is used, is made in the top CMake file by a parameter named "arch".

Platform Init The file plat_init.h defines a function plat_init(), which is responsible for the initialization of the CPU. This includes clocking, timers, interrupts, watchdog and so forth. It is recommended to divide those into further sub modules. The plat_init() function is called only at the beginning of the main function, so to say after a reset.

Timers timer.h contains prototypes of the functions get_actual_time(), plat_stop_timer() and plat_start_timer(), whereby get_actual_time() returns the actual system tick count, plat_stop_timer() stops the system tick timer and plat_start_timer() starts the system tick timer. The system tick count is represented by a global time_t variable named _sys_time. The content of the hardware counter register is added to _sys_time and set to zero before an overflow of the register can occur. This lets the system time always be represented as a 64 bit variable from the libraries point of view although the hardware timer register is smaller. get_actual_time() should return _sys_time plus the actual content of the hardware counter. In case the timer has 64 bit wide registers, it is not necessary to use _sys_time but instead return the actual register content. The tick frequency must be fixed at 1 MHz so that the resolution is 1 microsecond.

Interrupt The interrupt.h header file contains four function definitions namely enable_interrupt, disable_interrupt, get_interrupt_state and reset_device, where enable_interrupt enables the global interrupt, disable_interrupt disables it and get_interrupt_state returns its state. The reset_device function triggers a power on reset. These functions, except the reset, are used to create atomic sections.

GPIO The GPIO handling is defined in gpio.h. BMBS is designed to handle exactly four output pins, named in an enumeration called OUT_PIN, from PIN1 to PIN4. It is uncommitted which enumerated pin points to which physical pin. This is up to the programmer of the platform dependent code. The three functions set_pin_high, set_pin_low and toggle_pin do exactly what they are named after. It is recommended to use structures that suite the platform to further abstract the enumeration to a port and pin mapping. These structure for the pin mapping can then be used to create an array, with indexes corresponding to the enumeration.

Standard IO The std_io.h file defines the functions for the host - board communication. The plat_putc function sends exactly one char to the host while plat_getc gets exact one char from the host. Note that plat_getc is blocking. Beside this two functions, a receive interrupt, that arises whenever a char is received, must be assigned. Whenever this interrupt occurs, the function get_char_handler from the communication library (com.h) has to be called with the received char as an argument.

Linker Script Every platform implementation must contain a linker script for every supported toolchain. The naming of these files is based on the toolchain file. This linker scripts are commonly provided by the chip vendor but must be adapted for BMBS. There has to be a section named bmbs_benchmarks somewhere in the read only data area and two symbols which mark the begin and the end of this section. These symbols are __bmbs_benchmarksBegin and __bmbs_benchmarksEnd. bmbs_benchmarks acts as an array of structs and is located between __bmbs_benchmarksBegin and __bmbs_benchmarksEnd. This array like construct is filled up with structs containing information such as entry function and name during the build process. Furthermore a byte named bmbs_benchmark_count should indicate the number of benchmarks between __bmbs_benchmarksBegin and __bmbs_benchmarksEnd. This is done by subtracting __bmbs_benchmarksBegin from __bmbs_benchmarksEnd divided by the size of a single benchmark struct.

3.2 Benchmarks

Benchmarks that are grouped together are all placed in a separate subdirectory of the benchmarks directory. When CMake is beeing processed, one can choose to compile a subdirectory containing benchmarks against the build or not. This selection is achieved by a semicolon separated list named BENCHMARKS passed as an argument when calling *cmake*. It is possible to register more then one benchmark per subdirectory. They then are all listed separately from the host software.

3.3 Toolchain

The directory "toolchains" holds exactly one subdirectory for each platform. These subdirectories must be exactly named after the subdirectories from the arch directory. This is due to the fact that the directory names from the arch directory define the parameter names for the specific architecture and CMake only looks for a directory with this specific name. In this subdirectories there is one CMake file per toolchain, dedicated to compile the code for the selected platform. This CMake file defines the path to the cross compiler, compiler- and linker-flags, the path to the dedicated linker file, include directories for compiler dependent code, include directories for platform vendor libraries and the download target. The download target is the target to program the device. It contains some shell code to start the external programming tool and automatically downloads the binary file to the connected device. The selection which toolchain CMake file should be used is made in the top CMake file by setting the argument TOOL_CHAIN flag. If no explicit toolchain has been selected during the make build process gcc.cmake is used as default.

3.4 Library

The lib directory only contains platform independent code but relies on the implementations of the arch code. The functions defined and implemented in the lib directory are made for direct use in the benchmark code and are therefore referenced as include directory. If the programmer of a benchmark only uses the given functions from this library it is guaranteed that the code will compile and run on every platform. The four most important functions defined are tick, tock, xprintf and put_result.

Time Measurement tick and tock are used for time measuring. tick starts a measurement, wile tock delivers the time passed since tock was called. This is used to precisely measure the time consumed by a defined section between tick and tock.

Print Formatted xprinf is mainly used to inform the user about certain information from the benchmark and for debugging purposes. The messages printed with xprintf are displayed immediately to the user by the host tool.

Result Notification Put result is used for reporting time measurements to the user. There is a special result handling at the host software. This function must be used to report results instead of printing them with xprintf. Also the use of this function produces less overhead and is therefore much faster.

String Library Additionally there are some implementations of standard string functions e.g. strcpy, strcmp, strlen and so on. They could be used if the independence of the provided C library is desired.

Benchmark registration When programming a benchmark, the system has to be informed about the entry point of this benchmark and therefore a macro named BMBS_REGISTER_BENCHMARK was implemented. This macro creates an entry of a benchmark struct inside a list. For this the special bmbs_benchmarks section is exploited. The linker merges all structs containing the name, version and entry point into this section. bmbs_benchmarks then behaves like an array of this structs.

4 Build Process

The build process starts with calling CMake and with passing arguments for selecting the toolchain, the platform and a list of benchmarks. With this information CMake produces makefiles fitted to the platform and its needs. When processing this makefile via GNU MAKE all the components are compiled and linked together to a single file which could be programmed to the hardware. When calling CMake a list of supported platforms is created, which is a map of the sub directories from the arch directory. Afterwards this list is searched for the selected platform. If the respond is positive it calls the CMake file located in the regarding subdirectory to add all platform dependend code.
Then the toolchain directory is checked for a subdirectory with the exact same name in which a ⟨ toolchain name ⟩ .cmake file with the toolchain name given from the toolchain parameter must be present. This file tells CMake where to find the compiler, sets the compiler and linker flags, defines the platforms specific include directories and adds a download target for programming.
Now the benchmarks subdirectory is included. The CMake file in there checks the forwarded list of selected benchmarks and includes the corresponding benchmark directory or if the list is empty, all of them.
Following to this the lib directory is added to the project. The CMake file from lib is responsible for packing all the library source together to a link library which is then included to the build. At the end all collected files are compiled and linked together to a single file named bmbs.out as shown in Figure 3. This is the final binary.
The host tool is built by a separate Make file located in the host-tool directory.

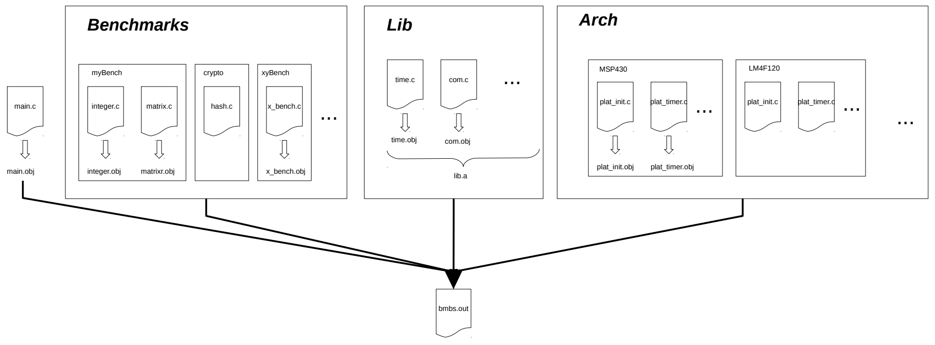


Fig. 3. Linking overview

5 Host - Device Communication

The host - device communication is build on a simple TLV (type length value) protocol transmitted over 8N1 UART. The first byte "Type" identifies the content of the data or describes an action that should be triggered e.g. start benchmark 0 or reset. The second byte indicates the number of data bytes and goes from 0x00 to 0xFE followed by the data itself (Figure 4). The length code 0xFF is used as a magic number and indicates a transfer of an unknown amount of data with zero termination, as shown in Figure 5. Sending a 0x00 as type indicates a protocol reset. This is useful if the synchronisation of host and device is lost, because 0x00 also terminates transmissions of unknown length. For synchronizing a series of zeros can be sent to bring the protocol state machine in a known state again. The following commands are implemented:

– Reset: Triggers a power on reset of the device.
– Start Benchmark: Starts the benchmark with the number given by the data byte zero.
– Echo: Sends back a echo response with the same data. This is for testing purpose only.
– Get Benchmarks: Returns the name and version of all benchmarks programmed on the device as a string of semicolon separated values.

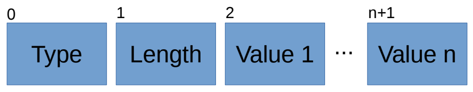


Fig. 4. Protocol with length

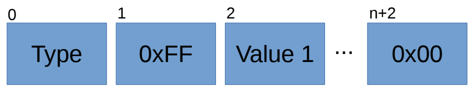


Fig. 5. Protocol with null termination

6 Evaluation

To evaluate the capabilities of BMBS three different algorithms with very varying demands and three different microcontrollers were implemented.

6.1 Platforms

Following platforms are used for evaluation:

- **ATmega328p**[2] is a mainstream 8 bit AVR architecture microcontroller from ATMEL. It is widely used especially by hobbyist. This Platform is also used by the Arduino Project [1].
- **MSP430F5529**[8] is a ultra low power 16 bit microcontroller from Texas Instruments. It is widely used in battery powered devices.
- **LM4F120**[7] is a ARM CortexM4F 32 bit microcontroller from Texas Instruments. ARM Cortex M4F are modern high performance mainstream CPUs.

6.2 Benchmarks

All benchmarks are compiled with the GCC from the corresponding platform. Optimization level is set to optimize for size (Os).

- **ATmega328p**: avr-gcc (GCC) 4.9.2
- **MSP430F5529**: msp430-elf-gcc (SOMNIUM Technologies Limited - msp430-gcc 6.2.1.16) 6.2.1 20161212
- **LM4F120**: arm-none-eabi-gcc (15:4.9.3+svn231177-1) 4.9.3 20150529 (pre-release)

- **matrix_mul** is a straight forward matrix multiplication based on array indexes and a more advanced implementation based on pure pointer arithmetic $A \cdot B$ with 8 bit, 16 bit and 32 bit matrices. The first matrix has a dimension of [10x5] and the second has a dimension of [5x10]. To save data memory the matrices A and B use the same data array and they also use the same data array for every bit size. When starting the benchmark the 8 bit matrix is initialized, a time measurement is started and $A \cdot B$ is calculated 1000 times with 8 bit numbers. Afterwards the time passed since the start of the measurement is divided by 1000 and reported back. This is done in the same way for 16 bit and 32 bit matrices with both implementations of the multiplication.
- **list_operations** is an implementation of a basic linked list. It implements following list operations:
  - insert
  - replace
  - delete
  - sort

  A list element consist of a pointer to the data and a pointer to the next element. At the beginning a data array is initialized with numbers starting with one to the number of elements which is in this case 50. After that a list of elements whose elements point to the previous initialized data is created. At this point the time measure starts and the elements are then connected to a list with alternating low and high numbers. The first operation sorts the list ascending. From this sorted list the odd elements are removed and in the next step replaced with the even elements. After this third operation the list is checked for the right data and length. The procedure after the start of the time measurement is repeated 100 times.
- **AES** provides arithmetic and table based implementations for AES128 and AES256. The benchmark is based on aes256_128 [6]. It first initializes a 128 byte buffer and a 256 bit key with data, starts the time measurement and encrypts the data buffer 100 times with the arithmetic implementation of AES128. Afterwards the time since the measurements start is taken, divided by 100 and reported back. The same procedure is done with the decryption afterwards. The buffer is then checked against his initial state to verify the correct encryption and decryption. AES256 and the table based versions of the algorithm are then tested in the same way.

With the matrix_mul example BMBS puts his great ability to compare different microcontrollers with the same algorithm in evidence. The result of such a comparison is shown in Figure 6 and Figure 7. Clearly recognizable is that a

32 bit architecture like the LM4F120 has no problem with 32 bit multiplication whilst the 8 bit ATmega328p struggles with it. Also interesting is that 8 bit multiplications with the straight forward array index based algorithm are faster on the ATmega328p than they are on the MSP430F5529 although the clocking frequency of the MSP430F5529 is 1.5 times higher then the clocking frequency of the ATmega328p. The low performance of the MSP430F5529 when processing the 8 bit multiplications for the array index based algorithm is due to the fact that indexes of the 16 bit and 32 bit multiplication are calculated by shift instruction while the array indexes of the 8 bit multiplication are calculated by multiplying numbers.

The AES example emphasizes the possibility to compare different implementations of an algorithm on the target microcontroller. Figure 8 shows the result of the AES benchmark on the LM4F120 microcontroller. It shows that arithmetic AES128 is only about 0.4 times faster than arithmetic AES256 but a table based AES256 is 62.6 times faster than arithmetic AES256.

With the list_operations example the efficiency of loops combined with pointer dereferencing can be checked. Figure 9 shows that the LM4F120 and the MSP430 are more or less equal with respect to their CPU frequency. Only the ATmega328p with his 8 bit architecture shows small weaknesses.

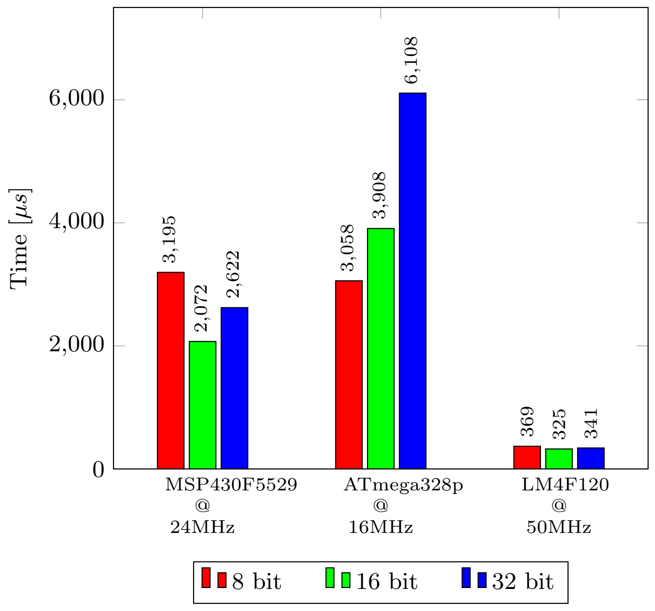


Fig. 6. Platform Comparison Example Of A [10x5] · [5x10] Matrix Multiplication With Array Indexes

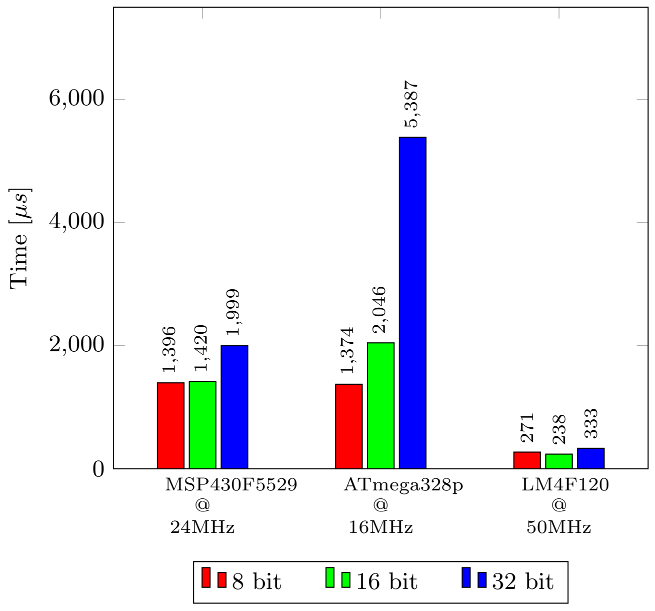


Fig. 7. Platform Comparison Example Of A [10x5] · [5x10] Matrix Multiplication With Pointer Arithmetic

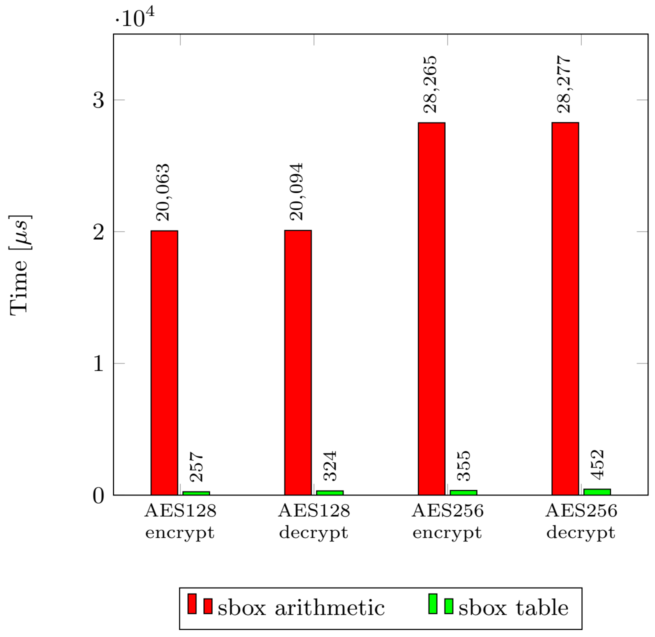


Fig. 8. Algorithm Comparison Example On LM4F120 @ 50MHz using AES benchmark

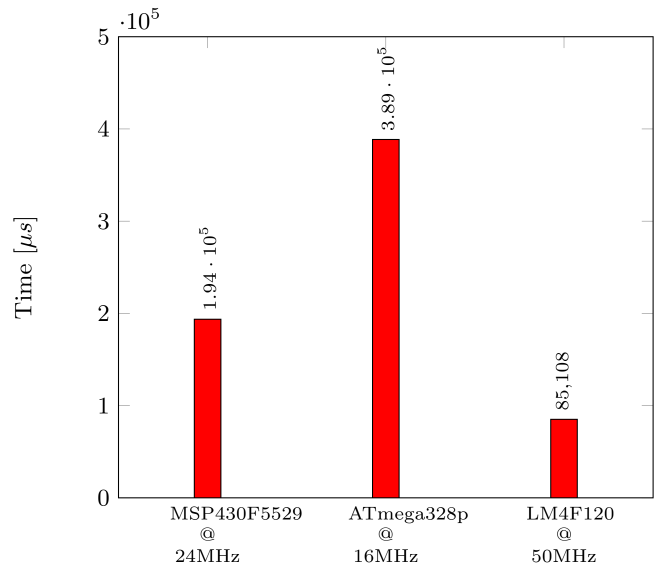


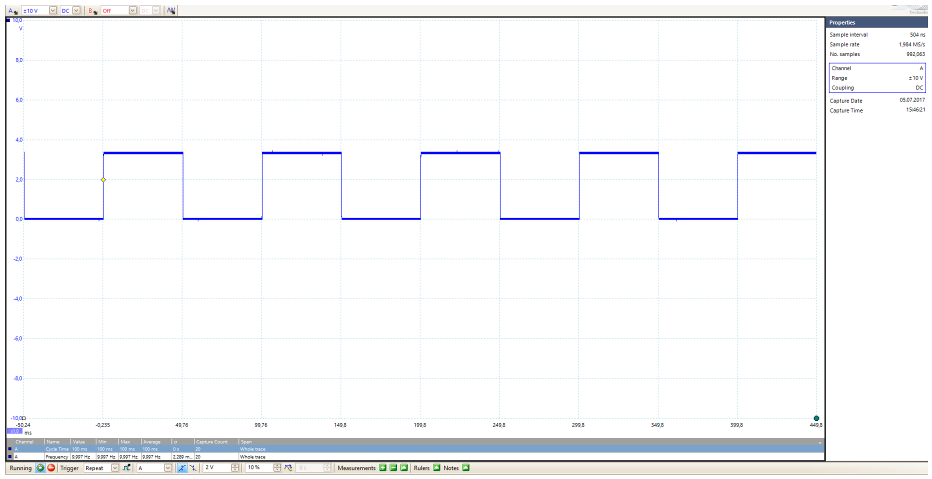
Fig. 9. List Operations On Different Platforms

6.3 Timing

To estimate the accuracy of the timing, a test benchmark that uses the GPIO and timer lib to produce a 10 Hz signal is implemented. This benchmarks contains a loop in which a time measurement is started and a second loop continuously checks the actual time against the time from measurement start plus 50000 μs. If the actual time is equal or bigger, an output pin is toggled and the outer loop continues. This signal was measured with a PicoScope5203 to check if the the frequency matches 10 Hz. The results for the three testing platforms are shown in Figure 10, Figure 11 and Figure 12. As shown in table 1, all the timings match the 10 Hz pretty accurate.

Table 1. Timing Errors

MCU	Frequency [Hz]	Error [%]
LM4F120	9.997	-0.03
MSP430F5529	9.998	-0.02
ATmega328p	10	0



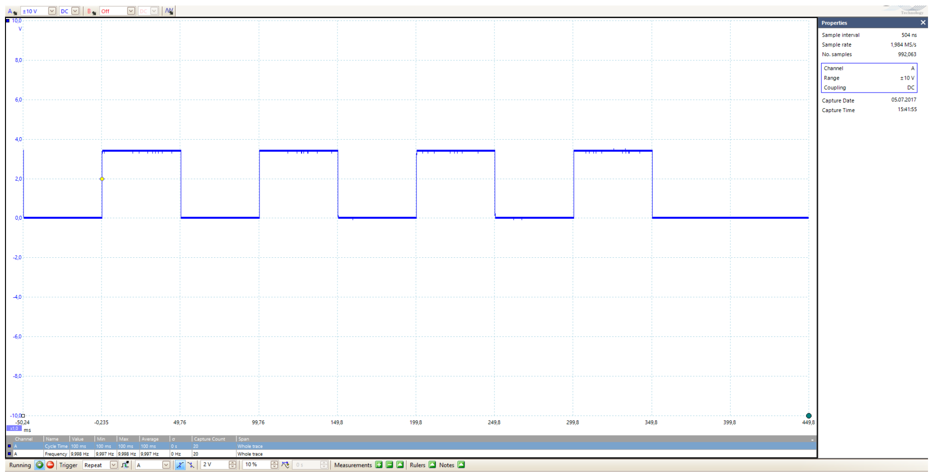
Properties

Sample interval	504 ns
Sample rate	1,984 MS/s
No. samples	992,063
Channel	A
Range	±10 V
Coupling	DC
Capture Date	05.07.2017
Capture Time	15:46:21

Channel	Name	Value	Min	Max	Average	σ	Capture Count	Span
A	Cycle Time	100 ms	100 ms	100 ms	100 ms	0 s	20	Whole trace
A	Frequency	9,997 Hz	9,997 Hz	9,997 Hz	9,997 Hz	2,289 m...	20	Whole trace

Running Trigger Repeat A 2 V 10 % 0 s Measurements Rulers Notes

Fig. 10. LM4F120 Timing measurement



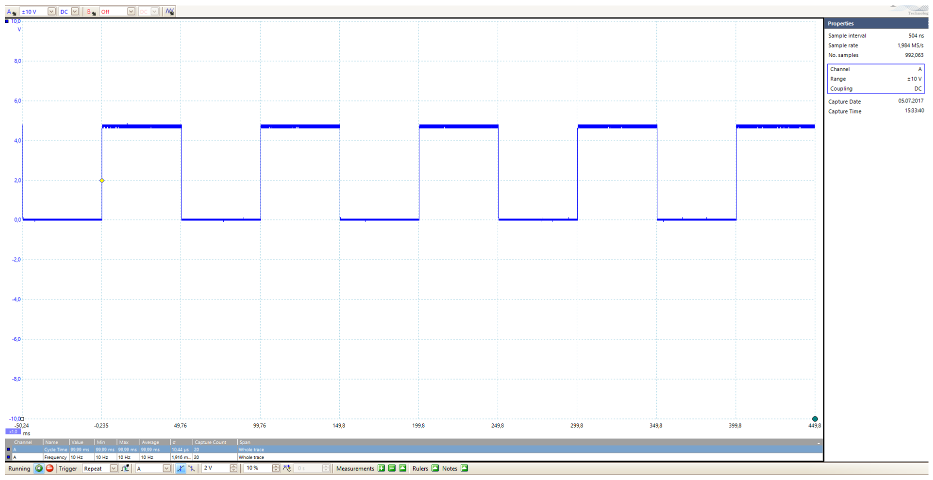
Properties

Sample interval	504 ns
Sample rate	1,984 MS/s
No. samples	992,063
Channel	A
Range	±10 V
Coupling	DC
Capture Date	05.07.2017
Capture Time	15:41:55

Channel	Name	Value	Min	Max	Average	σ	Capture Count	Span
A	Cycle Time	100 ms	100 ms	100 ms	100 ms	0 s	20	Whole trace
A	Frequency	9,998 Hz	9,997 Hz	9,998 Hz	9,997 Hz	0 Hz	20	Whole trace

Running Trigger Repeat A 2 V 10 % 0 s Measurements Rulers Notes

Fig. 11. MSP430 Timing measurement



Properties

Sample interval	504 ns
Sample rate	1,984 MS/s
No. samples	992,063
Channel	A
Range	±10 V
Coupling	DC
Capture Date	05.07.2017
Capture Time	15:33:40

Channel	Name	Value	Min	Max	Average	σ	Capture Count	Span
A	Cycle Time	99,99 ms	99,99 ms	99,99 ms	99,99 ms	10,44 µs	20	Whole trace
A	Frequency	10 Hz	10 Hz	10 Hz	10 Hz	1,916 m...	20	Whole trace

Fig. 12. ATmega328p Timing measurement

7 Conclusion

BMBS was designed as a benchmark suite for micro controllers. It makes it possible to run various benchmarks on various platforms. Attention was paid to have a strict separation of platform dependent and platform independent code. Therefore all benchmarks are platform independent and run on every implemented platform as long as there are sufficient resources like for example memory. BMBS showed its great ability to compare different algorithms on the same platform and the same algorithm on different platforms. Because it is not only a benchmark code but a whole suite, it is very comfortable to use. It selects the right platform depending code for the selected platform, cross-compiles and links it together to a single binary.

References

1. Arduino: arduino, `https://www.arduino.cc/`
2. ATMEL: atmega, `http://www.atmel.com/products/microcontrollers/avr/megaAVR.aspx`
3. Curnow, H.J., Wichmann, B.A.: A synthetic benchmark. Computer Journal 19(1), 43–49 (1976)
4. EEMBC: EEMBC coremark - processor benchmark, `http://www.eembc.org/coremark/about.php`

5. GNU: Gnu make, `https://www.gnu.org/software/make/manual/make.html/`
6. Ilya O. Levin, Hal Finney, P.S.: aes256_128, `https://github.com/pfalcon/aes256_128`
7. Instruments, T.: Lm4f120, `http://www.ti.com/tool/EK-LM4F120XL`
8. Instruments, T.: Msp430, `http://www.ti.com/lsds/ti/microcontrollers-16-bit-32-bit/msp/ultra-low-power/msp430f5x-msp430f6x/overview.page`
9. Kitware: Cmake overview, `https://cmake.org/overview/`
10. Weicker, R.P.: Dhrystone: a synthetic systems programming benchmark. Communications of the ACM 27(10), 1013–1030 (1984)