Compact Implementations of Shabal

Thomas Pornin, <thomas.pornin@cryptolog.com>

October 11, 2010

Abstract

We present here several implementations of the Shabal family of hash functions. These implementations have been optimized primarily for code size, for several architectures: x86 (32-bit and 64-bit), MIPS32, PowerPC (32-bit) and ARM (both with and without Thumb mode). Two portable C implementations are also provided. The implementations follow a streamable, thread-safe, reentrant API designed for immediate usability in any application. The compiled code size for those implementations, including the API, is well below one kilobyte, and as low as 404 bytes (on ARM-Thumb). The performance remains high: those compact implementations achieve at least 60% of the speed of the generic unrolled implementation of Shabal on the same systems[1].

1 Why Code Size Matters

The first and foremost processing performance characteristic which is measured for hash function implementations is raw speed, expressed in cycles per byte (for long messages) and in total number of clock cycles (for short messages). We hereby argue that code size is, in a substantial number of performance-critical applications, a criterion of at least equal importance.

1.1 Small Messages and Cache Locality

Modern CPU are much faster than their RAM; the typical latency for a RAM access is at least 15 ns, which equates to 40 or more clock cycles on a basic workstation. In order to

[1] This work was partially supported by the French Agence Nationale de la Recherche through the SAPHIR2 project under Contract ANR-08-VERS-014.

avoid a considerable slowdown, CPU use *caches*: small blocks of specialized RAM, with a much lower latency, which contain a local copy of parts of the main RAM. A cache is filled by small chunks, called "cache lines" (although RAM latency is high, it offers a high bandwidth for consecutive bytes). The innermost level of cache (called "L1") is usually split into separate caches for instructions and data. The instruction L1 cache size typically ranges from 8 to 64 kilobytes. If the implementation of the core loop of a given algorithm cannot fit in the instruction L1 cache, then the performance penalty is often severe.

When a hash function is benchmarked, the benchmarking framework normally consists in a tight loop which runs the hash function over the same block of data, with a number of iterations large enough that the total computing time can be measured with adequate accuracy. A few calls are even performed beforehand as a "warm-up", to make sure that the hash function code is already in the instruction L1 cache.

A benchmark should be representative of actual performance when the hash function is integrated in an application. The hidden assumption in the procedure described above is that in an actual application, the hash function implementation has exclusive use of the CPU caches. This assumption is valid when the processed message is "long" (more than a dozen kilobytes), thanks to *buffering*: the data which is to be hashed is accumulated in a temporary RAM zone (small enough to fit in data L1 cache, the traditional buffer size being 8 kilobytes), and the hash function implementation is invoked only when the buffer is full. When the function begins processing the buffer data, its code may have to be brought back from main RAM into L1 cache, and in doing so, some of the code of the rest of the application core loop may be evicted from cache; such code will have to be reloaded from RAM when the hash function implementation has finished processing the buffered data. However, such overhead may become negligible if the buffer is large enough: after a few dozen input bytes, the hash function implementation reaches its full speed. The benchmark thus uses an *approximation* of what really happens in an actual application, but the proper use of a buffer can make that approximation reasonably accurate.

This breaks down when considering small input messages: if the input data is very small, then the hash function has finished its work much before having reached its cache-optimal state. The speed measured by the benchmark will be meaningful only if the hash function implementation can reside in instruction L1 cache *along with* the rest of the application core loop. Whether this is easy or even possible depends on the rest of the application, and this is not captured by the usual benchmarks. However, a very compact hash function implementation makes it much more probable that it will fit. Alternatively, if the hash function must necessarily be reloaded from main RAM (e.g. because the application core loop itself overflows the L1 cache even without any hash function), then a hash function implementation with a very small footprint will be loaded faster, and lower overall RAM activity.

The code size effect on performance is most visible on architectures which have small L1 caches, and have low CPU power compared to their I/O capabilities. This points to

embedded systems, in particular networked boxes with 32-bit RISC CPU (MIPS, ARM, PowerPC...), as primary targets for code size optimization.

1.2 Code Size Contraints

There are situations where code size is under severe constraints. These situations are not commonly encountered on PC and workstations, but frequently occur in embedded systems. A few example are:

- *Smartcards and microcontrollers*: smartcards are very limited in just about everything: ROM, RAM, CPU power... ROM is often considered to be somewhat cheap (about six times cheaper than a RAM block of the same size), but must contain the complete application code and operating system. It is rarely practical to devote more than 3 or 4 kilobytes to a hash function implementation.
- *Boot code*: boot code is the code which is executed at system startup, before having access to the system storage units. It often comes from a dedicated ROM or Flash chip. Space is a scarce resource on such chips. In particular, systems which enforce a "trusted computing" framework will need to keep the boot code in a tamper resistant area (where size and power restrictions are most stringent), and in that situation the boot code will have to verify a MAC or a digital signature on any externally loaded code before using it, a process which is prone to entail a hash function computation.
- *Software updates*: some embedded systems can have their software contents externally updated, e.g. when first booted in factory (as a "customization" step) or after deployment in the field. The transfer medium may be slow (RS232 link, radio, phone line...), making code size an expensive commodity. This also applies to firmwares on USB devices: some devices require at boot time the upload of their application code, sent by the device driver from the host system.

Generally speaking, hardware vendors are much reluctant to integrate cryptographic algorithms, when the implementation size is on the order of ten kilobytes or more. A hash function implementation will be considered as mostly harmless, code-size wise, only if it fits in less than one kilobyte or so.

1.3 Code Size, SHA-3 and Shabal

At the time these lines are written, the SHA-3 competition[1] involves 14 "candidates" which are actively compared with each other, for security and performance. Raw speed,

in benchmark conditions, is estimated through the eBASH[2] (for PC-like hardware) and XBX[3] frameworks (for microcontrollers and embedded systems). Of these two, only XBX publishes code size measures. However, XBX reuses the same API than eBASH, which is designed for speed measures only. API and architecture call conventions are not important for bandwidth measures in benchmark conditions, but they have a non negligible effect on code size, especially for implementations which are optimized for low footprint.

We define a more generic API, consisting in a hash context structure definition, and three functions (for initializing the context, processing a chunk of data, and finalizing the hash computation). The API is meant for immediate applicability: an implementation following that API should be usable as-is in any application. API characteristics include the following:

- The API is *thread-safe* and *reentrant*, and implementations may be used in asynchronous contexts such as signal handlers.
- The API is *streamable*: arbitrary long messages can be processed by incremental chunks.
- The context captures the hash function state and can be used to clone the current computation (e.g. to get "the hash of input data so far").
- Implementations are *position-independent*: they can be used in DLL (on architectures that support DLL).
- Implementations are *portable*: the API uses basic C constructs and forbids uses of external functions except the standard `memcpy()` and `memset()` which are found even in very incomplete implementations of the C runtime.
- The API allows for simultaneous implementation of several hash functions within the same application, and in particular providing implementations of a SHA-3 candidate with the four standard output sizes (224, 256, 384 and 512 bits).

In Annex A, we detail the API and how exactly we measure the code size. This includes the definition of a number of target architectures; we define precise target models, ABI and implementation rules. Those rules are, by definition, arbitrary, but we tried to select systems which are representative of the current state of affairs in the world of embedded systems.

Following that API, we implemented the Shabal[4] hash function on all defined targets. The resulting code is published under a permissive, open-source license (actually the closest possible to Public Domain which still makes sense under French copyright law). Section 2 gives details on the implementation strategy, and lists code size and speed results. Variants of these implementations were also sent to XBX[3].

1.4 On RAM Usage

RAM usage is also an important performance criterion on some systems (especially small microcontrollers and smartcards). This is one of the reasons why we use a streamable API, where input data is provided as successive chunks: it allows hashing of messages much bigger than what could fit in RAM. We would like to point out that in the context of a streamable API, there are *two* distinct RAM usages to take into account:

- Some RAM is used for the duration of the hash computation; in our API, this is the context structure.
- Some RAM is used transiently, during the processing of a data chunk, but released immediately afterwards.

The "transient RAM" is likely to be less expensive in a typical application, since it can also be used (for local variables) by other code in the data processing path, in functions called from the same site than the hash function implementation.

On extremely limited systems (e.g. with less than 1 kilobyte of RAM), our streamable API implies non-negligible overhead (in particular, for a function with n-byte blocks, the context structure must have a n-byte buffer), and a C-based API does not make much sense on such architectures (C programming implying too much stack overhead).

1.5 Speed/Size Ratio

In order to try to capture the costs related to optimization for code size, we define the following speed/size ratio:

$$r = \frac{10^6}{cs}$$

where c is the processing speed on a long message, expressed in cycles per bytes, and s is the code size in bytes. A "long message" is an input message which is sufficiently long to make initialization and finalization costs negligible. The "10^6" value is just a normalization constant so that the numbers will be in a range adequate for human consumption.

This ratio is quite arbitrary. Speed/size ratios are common in the world of hardware implementations, in which a ten times smaller circuit may translate into ten implementations operating in parallel. The underlying assumption is that parallelism can be automatically applied, which, in the context of hashing data, is true only if there are several independent messages to hash, or at least a tree-like hashing mode. That assumption hardly applies to software.

2 Shabal Implementations

Here is a synthetic summary of the obtained code sizes and speed. Details on each architecture are given in the subsequent sections.

Architecture	**size**	**speed**		**speed/size**	**test system**
		long	short		
MIPS (LE)	856	42.1	25000	27.7	Broadcom BCM3302
MIPS (BE)	952	-	-	-	-
PowerPC (32-bit, BE)	768	16.4	6335	79.5	PowerPC 750 (G3)
ARM (ARM only, LE)	656	32.3	12392	47.2	Atmel AT91RM9200
ARM (mixed, LE)	536	32.6	12612	57.2	Atmel AT91RM9200
ARM (Thumb, LE)	404	47.0	18282	52.6	Atmel AT91RM9200
ARM (ARM only, BE)	712	-	-	-	-
ARM (mixed, BE)	576	-	-	-	-
ARM (Thumb, BE)	444	-	-	-	-
AVR8	908	181.7	68140	6.1	Atmel ATmega1281
i386	450	12.9	5237	172.1	Intel Core2 Q6600
amd64	416	13.0	5255	185.2	Intel Core2 Q6600

“LE” means “little-endian” while “BE” means “big-endian”. The big-endian MIPS and ARM implementations were tested under emulation only, with QEMU[5]; the QEMU emulator is not cycle-accurate, which is why we do not have speed measures on those systems. It is expected that the big-endian versions are 5% to 15% slower than the corresponding little-endian implementations on the same system (since our MIPS and ARM targets do not offer simple ways to perform cross-endian memory accesses, the big-endian implementations must “manually” byteswap the input words before applying the inner function).

A “long message” is a conceptually infinitely long message; the benchmark calls the `shabal()` function repeatedly over an 8-kilobyte buffer in a tight loop, so that the total running time exceeds 2 seconds. The speed is given in cycles per processed byte. A “short message” is a 32-byte message; the processing time is expressed in clock cycles. Please note that even if we give many significant digits, the benchmark accuracy is around 2%.

The “speed/size” ratio is defined in section 1.5.

For Shabal, all messages with length 0 to 511 bits are “short” and yield the same processing time. That time is very close to the speed on long messages, for 384 input bytes: this is what is theoretically expected (six invocations of the inner function, which processes 64 data bytes at a time), and it is verified in our measures. That time could be reduced by 33% by using a precalculated initial state, but this would enlarge code size by about 130

bytes (depending on the architecture) and would make the implementation specific to a single hash output size.

2.1 API

All our implementations follow the C API defined in the `"shabal_tiny.h"` file, as described in Annex A. Although the contents of the `shabal_context` structure are private to the implementation, they all use the structure similarly. For debugging purposes, here are the field significations:

`buf`
: The buffer for input bytes; processing is done by 64-byte blocks, and the last, incomplete block is stored in `buf`, beginning with offset 0.

`ptr`
: The current length of the data stored in `buf`; the current incomplete block uses `buf[0]` to `buf[ptr-1]`, inclusive. By construction, the value of `ptr` is always between 0 and 63 (inclusive).

`state`
: The current Shabal state, consisting of the A (twelve 32-bit words), B (sixteen words) and C (sixteen words) arrays, in that order.

`Wlow, Whigh`
: The current block counter W (low and high 32-bit halves). The counter assumes the value which will be used for the next block, when it is completed.

`out_size`
: The output size for this hash computation, expressed in bits (exception: in the AVR8 implementation, only the least significant byte of that field is set, and it contains the output size expressed in bytes).

All implementations simultaneously support all Shabal output sizes (all multiples of 32, from 32 to 512 bits, inclusive).

2.2 Portable C

We wrote two C implementations, called `portable` and `portable_lowram`. The code tries to use an unsigned integer type of exactly 32 bits, and on some systems which are recognized as using the little-endian convention, the data decoding is simplified, resulting in shorter and faster code.

`portable_lowram` is a rather direct transcription of the Shabal specification. The AVR8-specific implementation uses this C code as structural template.

`portable` is an optimization in the handling of the intermediate values for B. In the Shabal specification, in the main loop of the P permutation, values B_i, B_{i+6}, B_{i+9} and B_{i+13} are accessed at round i, where the index is to be computed modulo 16. The `portable` implementation stores the successive values of B in a temporary array; the values of the B words after addition of the message words and rotation by 17 bits are stored in the first 16 slots of the array, while the next 48 slots are used for the new B words which are computed in the main loop. The extra buffer has size 256 bytes (thus somewhat doubling RAM usage, albeit only in a transient way) but allows for easier access: the "modulo 16" operation is no longer needed. This makes the compiled code both smaller and faster. All the architecture-specific implementations, except the AVR8 code, follow that structure.

2.3 Architecture-Specific Assembly

We wrote assembly implementations of Shabal for a number of architectures. They all follow the same API than the portable C implementations, with the same context structure. We list here a few specific notes on each architecture.

2.3.1 MIPS

Architecture	**size**	**speed**		**speed/size**	**test system**
		long	short		
MIPS (LE)	856	42.1	25000	27.7	Broadcom BCM3302
MIPS (BE)	952	-	-	-	-

The test platform is a WRT54G Linksys WiFi router, running OpenWRT (a Linux variant). The CPU is a Broadcom BCM3302, clocked at 200 MHz. It follows the MIPS32 Release 1 architecture, but our implementation is actually compatible with MIPS I, down to the "load delay" rule. The overhead for MIPS I compatibility is very slight: 12 extra bytes of code, and no measurable speed difference.

Our code uses its own, reduced `memcpy()` implementation; this avoids the overhead of the MIPS position-independant calling convention, and allows for (optionally) using a very compact byte-by-byte memory copy code. Our function is 40 bytes larger than such a minimal copy function, but is able to perform copies by 32-bit words (using the `lwl`, `lwr`, `swl` and `swr` opcodes for unaligned access), which saves 2.5 cycles per byte.

The MIPS architecture has no rotation (this was added in MIPS32 Release 2); instead, we must use two shifts and one bitwise OR, which is expensive in code space (12 bytes of code per rotation).

The big-endian implementation includes an additional byteswapping step, which is applied on the input words right before calling the inner function, and also on the C words immediately before copying out the hash result.

2.4 PowerPC

Architecture	**size**	**speed**		**speed/size**	**test system**
		long	short		
PowerPC (32-bit, BE)	768	16.4	6335	79.5	PowerPC 750 (G3)

The test platform for PowerPC is a “clamshell” iBook from Apple, dating from year 1999. The CPU is a 300 MHz PowerPC 750, also known as a “G3”.

Our implementation uses `mulli` opcodes for the multiplications by 3 and 5; this is efficient in space (one opcode for the multiplication) and is also fast on the G3. On the more recent Cell and Xenon processors (used, respectively, in the PlayStation 3 and Xbox 360 consoles), the `mulli` instructions imply a high latency, and it would be worthwhile to replace them with shift-and-addition sequences.

The PowerPC is a big-endian processor but can perform cross-endian memory accesses with the `lwbrx` and `stwbrx` opcodes. It also tolerates unaligned memory accesses, which we use when producing the final hash result.

2.5 ARM

Architecture	**size**	**speed**		**speed/size**	**test system**
		long	short		
ARM (ARM only, LE)	656	32.3	12392	47.2	Atmel AT91RM9200
ARM (mixed, LE)	536	32.6	12612	57.2	Atmel AT91RM9200
ARM (Thumb, LE)	404	47.0	18282	52.6	Atmel AT91RM9200
ARM (ARM only, BE)	712	-	-	-	-
ARM (mixed, BE)	576	-	-	-	-
ARM (Thumb, BE)	444	-	-	-	-

We provide no less than six implementations for the ARM platform, owing to the wide variety of instruction sets and call conventions. The three little-endian versions were tested

on an Artila M501 board, and a Hewlett-Packard HP50g scientific calculator. Both use an ARM920T core (ARMv4T architecture), and yield the same speed measures. The big-endian versions were tested under QEMU emulation only; similarly to the MIPS platform, the ARM provides no easy way to perform cross-endian memory accesses[2], hence an extra byteswapping function is added, at a theoretical runtime overhead of about 5% to 10%.

Our implementations are compatible with both the old ATPCS and the new AAPCS call conventions; this comes naturally from the fact that the API functions use 32-bit integer types only. We only had to take care of keeping 8-byte stack alignment before calling `memcpy()`, and this incurred no extra code size cost.

The ARM-only and mixed ARM/Thumb implementations use the same inner function implementation (in ARM mode), hence the similar speed; the mixed implementation uses Thumb mode for the API functions, which are not timing-critical. The ARM core offers "free" rotations and shifts (for some opcodes, a rotation or a shift is systematically applied to one of the operands), which explains the good performance of the ARM with regards to the MIPS, even though both are supposed to execute one instruction per cycle. The ARM offers less general purpose registers than the MIPS (14 vs 31), but this turned out not to be an issue with Shabal. The Thumb-only code is more compact but slower: it does not benefit from the free shifts and rotations, and some operations must be split into several opcodes (notably because the 16-bit Thumb opcodes cannot embed immediate values as large as what the 32-bit ARM opcodes support).

2.6 AVR8

Architecture	**size**	**speed**		**speed/size**	**test system**
		long	short		
AVR8	908	181.7	68140	6.1	Atmel ATmega1281

The test platform is an ATmega1281 microcontroller. No physical board was involved: the code ran in the simulator integrated into AVR Studio 4, edited by Atmel. That simulator offers cycle-accurate timings and excellent simulation; the figures above are thus reliable.

In our implementation, a specific reduced `memcpy()` clone was used, because it allowed us to perform two pointer additions simultaneously (apart from copying data, it updates the source and destination pointers), and we can reliably invoke it with the `rcall` opcode which is smaller than the generic `call`.

The rotations in Shabal use bit counts of 1, 15 or 17 bits, and can be implemented with 1-bit rotations and some byte movements. This is very good for performance on the AVR8,

[2]This was added in ARMv6, with the `setend` opcode.

which offers only 1-bit shifts and rotations, and handles every data as individual bytes (8-bit registers).

The code size could be reduced further by mutualizing a few sequences of a dozen instructions in the inner function, as so many individual functions; but it would imply a non-negligible runtime overhead (a `rcall/ret` sequence needs 7 extra clock cycles).

While all our other assembly implementations offer between 60% and 75% of the speed of the fully unrolled C implementation of Shabal, our AVR8 assembly implementation turns out to be faster than all previously published AVR8 implementations of Shabal.

2.7 i386 and amd64

Architecture	**size**	**speed**		**speed/size**	**test system**
		long	short		
i386	450	12.9	5237	172.1	Intel Core2 Q6600
amd64	416	13.0	5255	185.2	Intel Core2 Q6600

The test system for speed measures is an Intel Core2 Q6600, clocked at 2.4 GHz. Only one core is used.

The x86 code comes in two versions, for 32-bit ("i386") and 64-bit ("amd64") modes.

The 64-bit code has access to eight extra registers (`r8` to `r15`) but it does not use them, because the encoding of opcodes which use the extra registers requires a one-byte additional prefix. The 32-bit code is thus an almost exact copy of the 64-bit code, with the same speed; what makes the 32-bit code larger is that it must comply with the i386 ELF calling conventions, where function arguments are passed on the stack instead of designated registers.

The 32-bit implementation is compatible with the original 80386 (it has no need for the i486 `bswap` opcode).

The x86 implementations call no external function at all (they inline their own versions of `memcpy()` and `memset()`), and are truly position-independant, with no relocation to handle by the dynamic linker.

3 Conclusion

We optimized Shabal for code size on a variety of architectures. The code size was always below 1 kilobyte, and as low as about 400 bytes for some architectures, while still providing an adequate speed. At the time of first publication, these implementations make Shabal the smallest SHA-3 candidate on all the target architectures we defined. This is not fully intrinsic: Shabal is very small, but it is expected that a few other SHA-3 candidates could be shrunk to similar code sizes, and possibly CubeHash16/32[6] could be made somewhat smaller. The Shabal implementations we present could also be shortened by a few dozen bytes on some architectures, but with a non-negligible extra runtime cost. Nevertheless, Shabal should still rank first, by far, for speed/size ratio measures on most platforms.

We encourage other implementers to try to optimize other hash functions for code size, in particular SHA-3 candidates. In the interest of making figures comparable, we recommend that such implementations follow the API and target architectures that we define in Annex A, or at least precisely document the API and implementation rules that they follow.

References

[1] *Cryptographic Algorithm Hash Competition*, `http://csrc.nist.gov/groups/ST/hash/sha-3/`

[2] *eBASH: ECRYPT Benchmarking of All Submitted Hashes*, `http://bench.cr.yp.to/ebash.html`

[3] *XBX: eXternal Benchmarking eXtension for the SUPERCOP Crypto Benchmarking Framework*, C. Wenzel-Benner and J. Gräf, Lecture Notes in Computing Science, Proceedings of CHES 2010, LNCS 6225, 2010, pp. 294–305. Web site on: `https://xbx.das-labor.org/`

[4] *Shabal, a Submission to NIST's Cryptographic Hash Algorithm Competition*, E. Bresson, A. Canteaut, B. Chevallier-Mames, C. Clavier, T. Fuhr, A. Gouget, T. Icart, J.-F. Misarsky, M. Naya-Plasencia, P. Paillier, T. Pornin, J.-R. Reinhard, C. Thuillet and M. Videau, `http://ehash.iaik.tugraz.at/uploads/6/6c/Shabal.pdf`

[5] *QEMU, open source processor emulator*, `http://wiki.qemu.org/`

[6] *CubeHash specification (2.B.1)*, D. J. Bernstein, `http://cubehash.cr.yp.to/submission2/spec.pdf`

A API and Implementation Rules

An *implementation* is a piece of code which follows the API and rules described below, and implements a hash function, or a family of hash functions. When several hash functions are implemented, the functions are defined by their output size. For instance, an implementation could provide code for both SHA-224 and SHA-256.

A.1 API

The API is defined as a C structure and three functions callable from application code written in C. The implementation contains a header file which declares the API elements; that header is meant to be included by application code which uses the implementation.

A *main identifier* is used to derive the names of the structure and the three functions. That identifier is free form, provided that it complies with the C rules for identifiers, is not reserved, and does not collide with a standard library name. For instance, it would be ill-advised to use "`malloc`" as identifier. In the examples below, we use the string "`xxx`" as a placeholder for that name.

A.1.1 Context Structure

The implementation shall define in its header a structure which serves as context for a hash function computation. The structure is given the type name "`xxx_context`" through a `typedef` declaration.

The structure contents are meant to be private to the implementation; however, they are defined in the header file (in C terminology, the type is *complete*) so that the application may allocate such structures, with the proper size and alignment.

The context structure shall follow these rules:

1. A structure instance contains the complete state for an ongoing hash computation. The application code shall be able to clone the state by simply copying the structure contents (e.g. with a `memcpy()` call). Note that this mostly implies that the structure must not contain any pointer to itself.
2. The implementation code must be able to handle any structure alignment that is compatible with the contents declared in the header file.

3. The structure contents are initialized with `xxx_init()`, updated by `xxx()` and used by `xxx_close()`. Before the first `xxx_init()` call, or after a `xxx_close()` and before the next `xxx_init()` call on the same context, the structure state is undefined. Only `xxx_init()` may be called with a context of undefined contents.

A typical context structure definition will look like this:

```
typedef struct {
        /* Here go the fields. */
} xxx_context;
```

A.1.2 API Functions

The implementation shall provide the three functions below, and declare their prototypes in the header file:

`void xxx_init(xxx_context *cc, unsigned out_size)`
: Initialize the context structure pointed to by "`cc`", for a new hash computation. The initial contents of the provided context structure are undefined, and, in particular, nothing guarantees that the context is initially full of zeros. This function shall fill the structure with an appropriate "initial state". The intended output size is given as the "`out_size`" parameter, expressed in bits. The implementation may assume that the output size is one of the sizes that it supports; if the implementation supports only a single output size, then it may ignore that parameter altogether.

`void xxx(xxx_context *cc, const void *data, size_t len)`
: Process some data bytes. The context structure, pointed to by "`cc`", must have been previously initialized with `xxx_init()`. The implementation may assume that the context is in a proper state. This function may be called an arbitrary amount of times on a given context, to provide data in several chunks.

  If "`len`" is zero, then the `xxx()` function does nothing; in that situation, it is acceptable that "`data`" is a null pointer. Otherwise, "`data`" shall point at a sequence of "`len`" readable bytes.

  The `xxx()` function may *not* assume that "`data`" follows any particular alignment.

`void xxx_close(xxx_context *cc, unsigned ub, unsigned n, void *dst)`
: Finalize a hash computation and produce the hash result. The context structure, pointed to by "`cc`", must have been previously initialized with `xxx_init()`.

  The "`ub`" and "`n`" parameters define 0 to 7 additional bits to append to the message data, as input through the `xxx()` function calls. "`n`" contains the number of extra

bits; the implementation may assume that the value of "n" lies between 0 and 7 (inclusive). The extra bits are in "ub", with numerical value 128, 64, 32, 16, 8, 4 and 2, in that order. For instance, to add the four bits "1101", "ub" shall have value 208 and "n" shall be equal to 4. The implementation may *not* assume that the non-input bits of "ub" are zero: the `xxx_close()` function must apply appropriate masking techniques. For instance, the four additional bits "1101" may also be provided with the numerical value 4565.

For the common case of an input message consisting in integral bytes only, then "n" is equal to 0 and this function shall ignore "ub". It is nonetheless required that the implementation supports non-zero values of "n".

It is possible that the "`xxx()`" function is not called at all on a given context, between the `xxx_init()` and the `xxx_close()` calls. In that case, the message to hash consists only in the extra bits in "ub". If "n" is also equal to 0, then the hash value corresponds to the empty message.

The hash value is written out in the buffer pointed to by "dst". This function may *not* assume that this pointer follows any particular alignment.

Once the `xxx_close()` has been called, the context structure contents are undefined. Only `xxx_init()` may be called again on that structure.

A.1.3 Header Layout

The header file shall have the following layout:

```
#ifndef xxx_h__
#define xxx_h__

#include <stddef.h>

typedef struct {
        /* The context contents */
} xxx_context;

void xxx_init(xxx_context *cc, unsigned out_size);

void xxx(xxx_context *cc, const void *data, size_t len);

void xxx_close(xxx_context *cc, unsigned ub, unsigned n, void *dst);

#endif
```

Please note the following:

- The header file is protected by a "guard macro" with a name derived from the main identifier. That macro makes the header file resilient to multiple inclusions from the application code within the same translation unit.
- The header file includes the standard header `<stddef.h>` in order to have a definition of the "`size_t`" type. For some very old compilers, dating from before the first C standard (published in 1989), it may be necessary to change this into `<stdio.h>` instead.
- The header file *may* include other standard headers (headers described in the ISO C standard), such as `<limits.h>` or `<stdint.h>`. It is better, in the interest of backward compatiblity, to use some preprocessor directives to avoid including a C99-only header (such as `<stdint.h>`) when compiling on a C90 ("ANSI C") system.
- The header should refrain from defining unnecessary types or macros, or declaring other functions. If it defines some extra elements, then those elements must have a name beginning with "`xxx_`", in order to avoid name collisions.

Most hash function implementations will need a 32-bit type. The code chunk below defines "`xxx_u32`" as an unsigned type of length *at least* 32 bits; on most systems it will have a length of *exactly* 32 bits. It can be included in the header file:

```
#include <limits.h>
#if defined __STDC__ && __STDC_VERSION__ >= 199901L
#include <stdint.h>
#ifdef UINT32_MAX
typedef uint32_t xxx_u32;
#else
typedef uint_fast32_t xxx_u32;
#endif
#else
#if ((UINT_MAX >> 11) >> 11) >= 0x3FF
typedef unsigned int xxx_u32;
#else
typedef unsigned long xxx_u32;
#endif
#endif
```

This code chunk highlights the rules explained above: the only new type defined by this code chunk has a name beginning with "`xxx_`" and the C99 header `<stdint.h>` is included only if a C99-compliant compiler is detected.

A.2 General Rules

A.2.1 Language and Files

Implementations may be written in any language. However, the implementation must ultimately compile into a single *object file* which exports external functions as defined by the C API, with the C calling conventions. In practice, this means that the implementation is written in C or assembly, or is written along a framework which can be automatically converted into C or assembly.

A.2.2 External References

The implementation may call only two external functions, which are described in the ISO C standard: `memcpy()` and `memset()`. These functions are very likely to be already included in any given application; thus, they can be considered as being "free" and their size is not included in the code size measure. All other external references are forbidden.

The GNU binutils tool "`nm`" can be used to list the symbol table of the object file. Here is a real example for a Shabal implementation on an ARM platform:

```
$ nm shabal_arm_mixed.o
000000d0 t $a
00000000 t $t
000000cc t inner
         U memcpy
00000085 T shabal
00000001 T shabal_close
000000d0 t shabal_inner
0000003d T shabal_init
```

This shows that the object files has only one external reference (marked with a "`U`"), to the `memcpy()` function. It also defines only three global symbols (marked with an uppercase "`T`" and they all begin with the "`shabal`" identifier.

Note: this rule might be slightly relaxed in the future, to include some common arithmetic functions on some platforms. For instance, multiplication or division routines on architectures which lack them.

A.2.3 Threads and Signals

The implementation shall be thread-safe and reentrant: two distinct threads may simultaneously access the implementation code, without any synchronization, as long as they operate on distinct context structures. This also extends to signal handlers.

In practice, this means that any global data table used by the implementation is read-only, and should be marked as such (e.g. with a "`const`" qualifier at the definition site, for C-based implementations).

A.2.4 Position-Independent Code

On those platforms which support it, the implementation shall follow the local rules for *position-independent code*. PIC code is code that can be included into a DLL (aka "shared library") because it does not contain absolute addresses. The ability to include the implementation code in a DLL is part of the general usability of this API.

Details on how to write PIC code depend on the architecture. Generally speaking, references to data tables and to external functions must go through an indirection table (the GOT, as "global offset table"), which is itself located relatively to the current instruction pointer. The dynamic linker fills the GOT, leaving code pages untouched (and thus amenable to sharing between processes, which is the point of the exercise). On most architectures (included all those currently defined for this specification), local jumps and local function calls use PC-relative opcodes and thus are inherently PIC.

On some platforms (e.g. PowerPC, ARM), calls to external functions can be made with a simple "local call" opcode; the static linker converts such calls into PIC-compatible calls with the use of automatically generated wrappers. Those wrappers are not counted in our size measures. On other architectures (x86, MIPS...), external function calls must use GOT-aware opcodes which are explicit at the assembly level.

C-based implementations need not worry about PIC: the C compiler does the appropriate job, when instructed to do so with a compile-time option.

On some platforms (e.g. PowerPC), there are two versions of the GOT-aware opcodes, depending on the size of the GOT, which itself depends on the number of functions defined and referenced from the complete shared library. The "short version" yields slightly smaller code, but is good only up to a given GOT size limit. When compiling C code with GCC, the difference between the two versions is the one embodied by the "`-fpic`" and "`-fPIC`" command-line options, instructing GCC to use, respectively, the short version or the long version. For the purposes of size measures, this specification mandates only the use of the "short version", because most applications using shared libraries will

nonetheless remain well below the GOT size limit.

A.2.5 Size Measure

The implementation code is first compiled or assembled into an object file. For C-based implementations, various optimization-related options may be set; however, the C compiler is always instructed to produce PIC code: with GCC, the "`-fpic`" command-line option is systematically used.

The total size of the object sections are then extracted and added. For object files following a format that GNU binutils can handle, the "`size`" tool is used. Here is a real example for a Shabal implementation on an ARM platform:

```
$ size shabal_arm_mixed.o
   text    data     bss     dec     hex filename
    536       0       0     536     218 shabal_arm_mixed.o
```

The total size is here 536 bytes, under the "`dec`" heading. The first three numbers are for the "`text`" (code and read-only data), "`data`" (read-write data with explicit initial values) and "`bss`" (read-write data initialized with zeros). If either of the "`data`" or "`bss`" section is non-empty, then you are doing something wrong: since the rules mandate that the implementation shall be safe with regards to concurrent access, even from asynchronous signal handlers, then all data shall be read-only, and thus placed in the "`text`" section. For a C-based implementation, presence of a non-empty "`data`" section means that there is a global table which lacks an appropriate "`const`" qualifier.

A.3 Architectures

This specification defines a number of test architectures, for which code size of a hash function implementation may be relevant. Some small details on, for instance, the target model within a given platform family, may impact the code size. Thus, we try to precisely define a set of architectures which should be used as a basis for comparing several implementations with each other.

When possible, the ELF binary format shall be used. ELF defines the call conventions used by the C programming language; it also specifies the rules for PIC code. Sometimes, several flavours of ELF exist; when applicable, we follow the one used by Linux derivatives on that platform.

Some architecture specifications mandate specific function prologues and exit sequences, and stack frame formats. These conventions allow for easier debugging. We do *not* insist on the strict observance of those rules, as long as "the code runs", including in the presence of asynchronous signals (please note that it implies that a valid stack must be maintained at all times) . They still should be observed on a general basis.

A.3.1 MIPS

The MIPS architecture family is one of the first commercial RISC architectures. The original architecture specification is known as "MIPS I". Later versions where named MIPS II, MIPS III, MIPS IV and MIPS V. MIPS III to V support both 32-bit and 64-bit computations and addressing, whereas MIPS I and II are 32-bit only. In 1999, two new variants were defined, called "MIPS32" and "MIPS64". MIPS32 is an enhancement over MIPS II, and is 32-bit only. The 1999 version of MIPS32 was renamed "MIPS32 Release 1" in 2002, when "MIPS32 Release 2" was published. A third release ("MIPS32 Release 3") was published in 2010 and also defines an optional instruction set called "microMIPS32", aimed at the production of more compact code.

Our target architecture is MIPS32 Release 1. This is the architecture that is commonly encountered in home routers and wireless access points, based upon custom chips by several vendors (Broadcom, Texas Instruments...). Also, the very popular PIC brand of microcontrollers by Microchip culminates in the PIC32, which is a MIPS32 core.

We specifically disallow the instructions introduced in MIPS32 Release 2, because many MIPS32-based systems do not support them. In particular, we do *not* allow the "`rotr`" or "`rotrv`" opcodes (32-bit rotations), nor the "`wsbh`" opcode (useful for endianness swap).

We follow the ELF specification for MIPS, as used by the OpenWRT operating system, an opensource Linux derivative for home routers. MIPS32 processors are nominally *endian-neutral*: they have a main endianness (selected at CPU reset time) used in supervisor and kernel mode, and each user mode process can optionally run in its own endianness. It is not possible for user mode code to swap its own endianness. Since endianness also impacts the encoding of the application opcodes themselves, the operating system chooses the endianness for each process and the process keeps it unchanged. OpenWRT tends to follow the little-endian convention, whereas older Linux instances on MIPS architectures used big-endian, as other historical Unix systems on that architecture. We therefore have two target systems:

- **MIPS32 Release 1, little-endian**
- **MIPS32 Release 1, big-endian**

the little-endian variant being slightly more in phase with the current trends.

Implementations should strive, if possible, at being compatible with the older MIPS I specification, in order to support older designs which may still be in use today. The additional following rules should be sufficient to ensure MIPS I compatibility:

- The MIPS I does not feature the "`movn`" and "`movt`" conditional move instructions.
- The MIPS I has the *load delay slot* rule: a memory read opcode (such as "`lw`") using a given register `r` as destination operand shall not be immediately followed by an opcode using `r` as source operand. In MIPS32, such a situation implies only a mostly harmless delay, but in MIPS I it may have undefined consequences; in particular, the second opcode might or might not use the previous value of `r` instead of the read word.

Other rules for MIPS I compatibility are about floating point and system interaction, and should be irrelevant to hash function implementation. MIPS I compatibility may slightly enlarge an implementation, but usually by a very small amount only.

Documentation:

- The *MIPS32® Architecture for Programmers* volumes can be downloaded from the dedicated MIPS Web site:

  `http://www.mips.com/products/architectures/mips32/`

  Volumes I (introduction) and II (instruction set) are useful for hash function implementation.
- The MIPS32 ELF ABI (*System V Application Binary Interface, MIPS® RISC Processor Supplement, 3rd Edition*), as used in Linux, can be downloaded from:

  `http://math-atlas.sourceforge.net/devel/assembly/mipsabi32.pdf`

A.3.2 PowerPC

The PowerPC architecture was derived in the early nineties from IBM's POWER platform. It was initially meant for desktop systems, and the Macintosh line of computers from Apple used them until they switched to Intel x86 architecture in 2005. As of 2010, the PowerPC architecture dominates the video game console market, since it is used as main CPU by the three major consoles (PlayStation 3, Wii and Xbox 360).

The PowerPC architecture is defined by the Power.org standards body, an emanation of the former IBM-Freescale alliance, and under nominal IEEE governance. The full-blown PowerPC architecture contains both 32-bit and 64-bit computations and addressing; however, many implementations of the PowerPC architecture use only the 32-bit subset. The first fully compliant PowerPC core, the 603e, is still licensed as of 2010, and massively produced by several licensees such as Freescale. The PowerPC systems enjoy a substantial market share in the automotive and aeronautics industries, notably because of a long history of hardened and shielded implementations. The RAD6000 and RAD750, based on PowerPC 603e and 750 respectively, are radiation-shielded single board computers which equip many of the current NASA space probes. The 66 Iridium satellites (for the satellite phone system of the same name) run with 603e cores.

The PowerPC can run in little-endian and big-endian modes, and this is programmatically adjustable. However, some operations are not possible in little-endian mode (e.g. the "`stmw`" opcode, which writes in memory the contents of several registers), so it can be said that the PowerPC is more comfortably used in big-endian mode. Almost all PowerPC-based systems use it in big-endian mode. At the assembly level, some opcodes are available, which allow for endian-swapped memory read and writes ("`lwbrx`" and "`stwbrx`").

We therefore define our target architecture to be the **PowerPC 603e in big-endian mode**. This implies 32-bit only registers. We use the ELF format and call conventions.

ELF states that the register `r2` is "reserved by the operating system" and `r13` is used for the "small data area". In practice, under Linux, these registers may be used for support of position-independent code and thread-local storage. A hash function implementation following this specification should not in general use thread-local storage, but it may be interrupted by a signal handler which does. Hence, the `r2` and `r13` registers shall be kept untouched at all times; it is not sufficient to merely save them on the stack and restore them later on.

Documentation:

- The *Power Instruction Set Architecture, Version 2.06* can be obtained from the Web site of Power.org, the entity which now standardizes the PowerPC architecture:

  `http://www.power.org/resources/downloads/`

- An alternate description of the 32-bit PowerPC instruction set can be found in the *Programming Environments Manual for 32-bit Implementations of the PowerPC™ Architecture*. This document covers 32-bit instructions only, and is somewhat easier to read than the bulky Power ISA:

  `http://www.freescale.com/files/product/doc/MPCFPE32B.pdf`

- The PowerPC 32-bit ELF ABI (*System V Application Binary Interface, PowerPC Processor Supplement*), as used in Linux, can be downloaded from:

  `http://refspecs.freestandards.org/elf/elfspec_ppc.pdf`

A.3.3 ARM

The ARM architecture was started in 1983, with the first ARM processor produced in 1985. ARM processors were initially meant for very efficient handling of hardware interrupts, but it soon turned out that the "small RISC" design of the ARM processor made it very appropriate for embedded systems, in particular mobile system, due to its very low power consumption. ARM cores are licensed to various constructors, and are often embedded in custom ASIC or FPGA designs.

Architectures and families ARM *architectures* describe the capabilities of the devices (in particular, instruction sets), while ARM *families* relate to the core implementations, which are declined into several variants. The architectures are designated with a lowercase "v", a number, and some optional letters. For instance, "ARMv4T" means "architecture version 4, with the Thumb instruction set". On the other hand, families have constructor specific names; the family names from ARM Holdings tend to begin with "ARM" but do not have the lowercase "v". The terminology is confusing at times; for instance, the ARM7TDMI family includes the ARM7TDMI processor, which implements the ARMv4T architecture. The ARM9 family includes several processors, some implementing ARMv4T (e.g. ARM920T) while others comply to the ARMv5T architecture, or a variant thereof (e.g. ARM926EJ-S).

Endianness ARM processors are usually *endian-neutral* and endianness can be configured at runtime[3]. However, the external memory representation is unchanged: a given 32-bit value is always stored in the same way, regardless of the current endianness setting in the processor. The endianness setting is thus mostly an interpretation of pointers to individual bytes. An equivalent way to state it is the following: when the processor endianness is switched, all 32-bit words in RAM are simultaneously byte-swapped. Consequently, the software endianness switch cannot be used to improve cross-endian decoding of data. As the ARM architecture reference manual puts it: "there is no point in changing the configured endianness of an ARM processor to be different from that of the memory system it is attached to, because no additional architecturally defined operations become available as a result of doing so."

[3] Some ARM implementations may be limited to only one endianness.

In later revisions (ARMv6), an endian-swap flag for memory accesses has been added, with a specific opcode to alter that flag (`setend`). This is not available in all ARM processors, even new ones, and we do not allow it.

Big-endian and little-endian ARM systems have been in use; the current trend in the Linux world seems to favour little-endian convention.

Thumb code Beginning with the ARMv4T architecture, ARM processors support a new instruction set called *Thumb*. In Thumb mode, instructions are simpler but shorter (16 bits per opcode, instead of 32 bits), allowing for more compact code. A later revision added 32-bit instructions to Thumb, yielding *Thumb-2*, which should combine compactness of Thumb code, and speed of the "normal" ARM instructions. Some new designs support only Thumb-2, or even only Thumb (for the Cortex-M0).

In processors which support both the original ARM instruction set, and the Thumb instruction set, both types of code can be mixed, provided that specific code sequences are used to switch between states. Code which uses those sequences (mainly using the `bx` opcode when calling a function through a pointer, or when returning to a caller) is said to support *Thumb interworking*.

Unfortunately, the `bx` opcode is not available on pre-ARMv4T processors, in particular the StrongARM family, which has been quite popular due to its high performance (when the first ARMv4T processors became available, they could be clocked up to about 40 MHz, while the contemporary StrongARM ranged up to 206 MHz with similar instruction timings). Making code which both supports Thumb interworking and runs on a StrongARM is possible, but tricky.

ABI Linux systems on ARM first followed the APCS call convention, later expanded into the ATPCS when Thumb mode was added.

Then ARM defined a new ABI, called AAPCS, which solves a few shortcomings of ATPCS, mostly related to floating point. The new ABI is also called "EABI", the previous one being renamed "OABI". Some Linux distributions switched to AAPCS, in particular the influential Debian system, on late 2007. The AAPCS is there handled as a whole new architecture; the name "armel" was coined, for "ARM, little-endian, using the AAPCS", while "arm" means "ARM, little-endian, using the ATPCS". The name "armeb" remains ambiguous since it was used for big-endian ARM with ATPCS and should be used for big-endian ARM with AAPCS too.

For the implementation of hash functions, floating point and system interaction are irrelevant; the remaining differences between ATPCS and AAPCS are mostly:

- AAPCS mandates 8-byte alignment of 64-bit integer types, both for structure fields and for register numbering;
- AAPCS mandates 8-byte stack alignment when calling public functions.

Since the API we describe here defines functions with 32-bit arguments only (on ARM), only stack alignment must be checked, and only when calling an external function such as `memcpy()`.

AAPCS also mandates Thumb interworking. Strictly speaking, it is not possible to comply to AAPCS while running on pre-ARMv4T architectures.

In both ATPCS and AAPCS, the `r9` register, also known as `v6`, is reserved by the operating system. In practice, it is used to implement thread-local storage. Since it may be used with these semantics from a signal handler interrupting the hash function implementation at any moment, it must not be altered, even transiently. Saving and restoring it is not sufficient.

Target Our main target system is the ARM7TDMI (of the family with the same name). This is a best-seller, which has been included in many custom ASIC, thanks to its very small size, low power consumption and heat generation, and its support for the Thumb instruction set, which allows for compact code. The ARM7TDMI is still widely used and can be licensed from ARM Holdings, although ARM9 cores are now recommended for new designs. The smaller ARM9 cores implement the same ARMv4T architecture than the ARM7TDMI, and are usually both faster and more power efficient.

We thus define six ARM targets, all using the ARM7TDMI and the ARM920T as referent platforms:

- **ARMv4, little-endian, ARM instructions only** (no Thumb);
- **ARMv4T, little-endian, ARM and Thumb instructions allowed**;
- **ARMv4T, little-endian, Thumb instructions only**;
- **the three same architectures, but in big-endian mode**.

The ARM-only versions shall target compatibility with a StrongARM, hence without Thumb interworking. The other versions shall support Thumb interworking, and assume that the platform provides an interworking compliant system library (thus making `memcpy()` callable from Thumb code).

Compatibility Layer It is possible to run Thumb-powered code on a Thumb-able processor with a non-Thumb-interworking system library, by adding some explicit mode-jump wrappers. This entails adding before each public function implemented in Thumb instruction a 8-byte header, here shown for the implementation of the `xxx_init()` function:

```
xxx_init:
        .code  32
        add    r12, pc, #1
        bx     r12
        .code  16
        @ here comes the Thumb function implementation
```

Similarly, calling an external ARM-only, non-interworking `memcpy()` from Thumb code will look like this:

```
        add    a4, pc, #0
        bx     a4
        .code  32
        bl     memcpy
        add    r12, pc, #1
        bx     r12
        .code  16
```

with two Thumb instructions (4 bytes) to switch to ARM code, and two ARM instructions (8 bytes) to switch back afterwards. Note the use of the `a4` register, which is not preserved by `memcpy()`, but is not used either as parameter (`memcpy()` has only three parameters).

Such wrappers are handy for testing on operating systems which follow the old ATPCS. However, we do not include them in our size measures. We assume that on an actual system where code size is important, with a Thumb-capable processor, Thumb code and AAPCS would be used.

Documentation:

- The *ARM Architecture Reference Manual* is available from ARM themselves, as a PDF, after free registration:

  `http://infocenter.arm.com/`

  (copies of various versions of that document can be found in many places on the Web).

- The ATPCS (*ARM-Thumb Procedure Call Standard*) and AAPCS (*Procedure Call Standard for the ARM® Architecture*) can also be found on that site:

`http://infocenter.arm.com/help/topic/com.arm.doc.espc0002/ATPCS.pdf`
`http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf`

A.3.4 AVR8

The AVR8 architecture is a line of 8-bit microcontrollers produced by Atmel. They are available in various configurations, depending on RAM size, ROM size (normally implemented as Flash), and computing capabilities. From an architecture point of view, Atmel designed three main cores:

- the *minimal core*, for the smallest devices;
- the *classic core*, with support for larger RAM and better memory access (e.g. opcodes for stack handling);
- the *enhanced core*, for larger applications, and with multiplier circuitry.

The classic and enhanced cores were also declined in versions with additional opcodes and registers to support more than 8 kilobytes of code.

The minimal core is not meant for C programming; for instance, it lacks hardware support for a stack, and can use only a small amount of RAM (256 bytes). GCC does not support the minimal core. The C-based API which we define here makes little sense on a minimal core.

Almost all the recent classic cores designed by Atmel also support the `movw` opcode, an import from the enhanced core. This is a handy opcode which can move two 8-bit registers at a time. It can be mechanically replaced by a pair of `mov` opcodes if compatibility with older classic cores is needed.

Our target is then the **AVR8 classic core with movw**. The "ATmega" series is representative of that target; XBX[3] uses an ATmega 1281, which is actually an enhanced core, but should provide timings similar to any other ATmega. Compatibility with a classic core with `movw` means mostly avoiding the multiplier instructions (`mul`, `fmul`...) and the opcodes for 24-bit address support (`eijmp`, `eicall`). With GCC and GNU binutils, the "`-mmcu=avr25`" command-line flag can be used to enable support for `movw`, and report as error any use of a more advanced instruction.

The ABI used by the compiler is documented in the AVR Libc FAQ. A summary is the following:

- The AVR8 architecture has 32 registers, `r0` to `r31`. Registers contain 8-bit values. Register pairs `r26` and `r27`, `r28` and `r29`, and `r30` and `r31`, are also interpreted as 16-bit registers (dubbed, respectively, `X`, `Y` and `Z`) when used in memory access opcodes, for addressing. In a register pair, little-endian convention is used: the lowest-numbered register contains the least significant byte.

- A called function must preserve registers `r1` to `r17`, `r28` and `r29`. It may modify them provided that it restores them upon exit. The hardware stack (accessed with `push` and `pop` opcodes) is handy for that. All other registers can be freely modified without having to restore their value.

- C compiled functions expect `r1` to contain the value zero upon entry. Thus, publicly callable functions may expect that value in `r1`, and must enforce it when calling other public functions.

- `int`, `size_t` and pointer types are 16-bit.

- Parameters are passed on registers `r25` to `r8`, in that order, from left to right. Parameters use register in *pairs*, even one-byte parameters, so that each parameter begins on an even-numbered register. For instance, the `xxx()` function will receive the pointer to the context structure in `r24-r25`, the pointer to the input data in `r22-r23`, and the data length in `r20-r21`. Little-endian convention is used for multibyte values. Parameters which do not fit in the 18 dedicated registers are passed on the stack, but this situation does not arise with our API.

There is no notion of position-independent code in AVR8, since there is no DLL. Functions can be called with `call` or `rcall`, the latter being more space-efficient (16-bit opcode, instead of 32-bit) but limited to calling functions within a $\pm$4 kilobytes code range. For the purposes of this API, we may assume that `memcpy()` and `memset()` are within that range, thus callable with `rcall`.

The AVR8 is a Harvard architecture, with separate address spaces for data and code. RAM is a scarce resource, hence read-only tables shall be placed in code space. Access to code space as data requires some specific opcodes (`lpm`). GCC and AVR Libc provide qualifiers and access macros to define and read such tables from C.

Documentation and tools:

- The *8-bit AVR Instruction Set* can be downloaded from the Atmel Web site:

  `http://www.atmel.com/`

- *WinAVR* is a port of GCC and GNU binutils as cross-compilers for the AVR platform, along with an AVR-specific implementation of the standard C library called *AVR Libc*. WinAVR runs on Windows systems. It can be downloaded from:

`http://winavr.sourceforge.net/`

- Also available from the Atmel Web site, after free registration, is *AVR Studio 4*, a development IDE which includes an editor, an assembler and a cycle-accurate simulator. AVR Studio automatically uses WinAVR if the latter is also installed; AVR Studio then offers the possibility to develop "GCC projects".
- The *AVR Libc* has its own Web site, with much information:

`http://www.nongnu.org/avr-libc/`

A.3.5 i386 and amd64

The main CPU of a typical PC is a large platform, inserted into a relatively expensive board. Code size constraints are unlikely to be much a problem; similarly, they offer instruction cache sizes of 32 kilobytes or more. Moreover, such platforms exhibit high computing abilities with regards to memory bandwidth, and data processing tasks are often more limited by I/O than hash function performance.

However, the 32-bit instruction set has been used for some SoC (Systems on Chip), e.g. the Vortex86, currently produced by DM&P Electronics. The Vortex86SX is a low-end, power-efficient clone of the Intel 80486SX. In particular, it has no FPU or vector unit such as MMX; it offers 32-bit integer arithmetics only.

Our 32-bit target is then the **Intel 80486SX**, because it is the defining reference for the 32-bit x86 instruction set that will be found in x86-compatible processors. For the purpose of hash function implementation, the main difference with the original 80386 is that the 80486 offers a byteswapping opcode (`bswap`) which is a great asset for implementing hash functions which follow the big-endian convention. For little-endian functions, there is little need for `bswap`, and implementations shall be compatible with the basic 80386.

The 32-bit ELF standard for Linux describes simple call conventions, which we follow. Position-independent code is achieved by computing the code position at runtime, and then using that address to locate the GOT. As we insist for DLL-compatible code, this procedure must be followed to call `memcpy()` and `memset()`. It is expected, however, that implementers will find it more space-efficient to simply inline memory copying routines with the "`rep movsb`" opcode.

For completeness, we define as secondary target the **amd64 architecture**, which is the "64-bit" mode of modern x86 processors. All amd64-compatible processors offer sixteen general-purpose 64-bit registers, as well as a 80387-compatible FPU, a MMX unit, and a SSE2 unit. An amd64-compliant processor is necessarily kind of huge, but (relatively) low power versions begin to become available, for "big mobile systems" such as netbooks.

The 64-bit ELF standard for Linux describes a call convention where the first arguments are passed in registers instead of the stack, which turns out to be more space efficient than the 32-bit call convention.

In the ELF conventions, unaligned memory accesses are tolerated, with at worst a minor timing penalty (at least for general purpose registers; some SSE2 opcodes have stricter rules). A little known feature of the x86 processors is that they can trap unaligned accesses, resulting in immediate process termination, if instructed so; this is activated by setting the `AC` flag (available since the 80486). Usual x86 operating systems do not do that, and the hash function implementation may assume that the `AC` flag is cleared.

For space and speed optimization, the following may be noted:

- In 64-bit code, 32-bit operations are considered slightly more "natural" in the following sense: a 64-bit operation is encoded as a 32-bit operation with an extra one-byte opcode prefix. Using 32-bit operations thus saves a little code space. Nonetheless, we insist on full API support; in particular, the "`len`" parameter to the `xxx()` function cannot be assumed to be less than 2^{32}. Implementations shall support hash inputs up to 2^{64} bits at least.
- In 64-bit code, eight extra registers are available, in addition to the registers already present in 32-bit mode. Their use can help avoid expensive storage operations into RAM (especially in code size: addressing modes with a stack pointer relative displacement imply huge encodings). However, any opcode using one of the eight "extra" registers will need a one-byte additional prefix, when compared to the same opcode using only the eight "traditional" registers.
- In both 32-bit and 64-bit code, the low byte of some of the registers can be addressed directly. This can be used to save some space (e.g. using an 8-bit instead of 32-bit constant). But mixing 8-bit and 32-bit accesses to the same registers appears to have adverse effects on speed. This should be done with caution.

Documentation:

- The *AMD64 Architecture Programmer's Manual*, volumes 1 (*Application Programming*) and 3 (*General Purpose and System Instructions*) detail the amd64 architecture:

  `http://support.amd.com/us/Processor_TechDocs/24592.pdf`
  `http://support.amd.com/us/Processor_TechDocs/24594.pdf`

- Reference manuals for the x86 architecture (32-bit and 64-bit) can be obtained from Intel's Web site:

`http://www.intel.com/`

However, since the x86 architecture is complex and we want to concentrate on the subset which was already present in the original 80386 and 80486, it may be easier to begin with the reference manual for the 80386, a copy of which being available on:

`http://microsym.com/editor/assets/386intel.pdf`

- The i386 ELF ABI (*System V Application Binary Interface, Intel386™ Processor Supplement*) can be found there:

  `http://www.sco.com/developers/devspecs/abi386-4.pdf`

  while the 64-bit version is available as:

  `http://www.x86-64.org/documentation/abi.pdf`