The ARM-THUMB Procedure Call Standard

Development Systems Business Unit

Engineering Software Group

Document number:	SWS ESPC 0002 B-01
Date of Issue:	24 October, 2000
Author:	-
Authorized by:	

Abstract

This document defines a family of procedure call standards for the ARM and THUMB instruction sets.

Keywords

procedure call, function call, calling conventions

Distribution list

Name	Function	Name	Function

Contents

1 ABOUT THIS DOCUMENT

1.1 Change control

1.1.1 Current status and anticipated changes

Release A-05 of this specification is the *first public release*, the ADS-1.0 release.

Release B-01 is the ADS-1.1 release. The description of parameter passing has been extensively revised, but where release A-05 can be implemented without major difficulties, the resulting parameter lists are almost always identical. Differences occur only where release A-05 proved too difficult to implement precisely.

Future releases of this standard are likely to:

- Define more precisely the interaction between this standard and the IEEE754 standard for floating-point arithmetic.
- Define more precisely the mapping of C, C++, and Java source language entities to the machine-level values passed as parameters across publicly visible interfaces.

1.1.2 Change history

Issue	Date	By	Change
A-01	18 March, 1998	-	First partial draft for early review (WD, HM, SW, EN)
A-01	20 March, 1998	-	Second partial draft for internal review.
A-02	30 March, 1998	-	First external-review DRAFT.
A-03	17 July, 1998	-	Second external-review DRAFT
A-04	23 October 1998	-	FINAL internal review DRAFT
A-05	5 November, 1998	-	Editorial changes following review
B-01	24 October, 2000	-	Added 8-byte stack alignment motivated by ARM architecture 5. Revised floating-point argument marshalling after it was found impossible to implement. Made minor editorial improvements.

1.2 References

This document refers to the following documents.

Ref	Doc No	Author(s)	Title

1.3 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	**Meaning**
PCS	Procedure Call Standard
APCS	ARM Procedure Call Standard
TPCS	Thumb Procedure Call Standard
ATPCS	ARM-Thumb Procedure Call Standard
Subroutine, routine	A fragment of program to which control can be transferred that, on completing its task, returns control to its caller at an instruction following the call.
Procedure	A routine that returns no result value.
Function	A routine that returns a result value. A C/C++ function.
Memory state	The state of the program’s memory, including values in machine registers.
Externally visible [interface]	[An interface] between separately compiled or separately assembled routines.
Activation (call-frame) stack	The stack of routine activation records (call frames).
Variable register, v-register	A register used to hold the value of a variable (usually one local to a routine).
Scratch register, temporary register	A register used to hold an intermediate value during a calculation (usually, such values are not named in the program source and have a limited lifetime).
Activation record, call frame	The memory used by a routine for saving registers and holding local variables (usually allocated on a stack, once per activation of the routine).
Parameter	A formal parameter of a routine given the value of the actual parameter when the routine is called.
Argument	Formal parameter or actual parameter according to context.
PIC, PID	Position-independent code, position-independent data.

More specific terminology is defined when it is first used.

2 SCOPE

This standard defines how subroutines can be separately written, separately compiled, and separately assembled to work together. It describes a contract between a calling routine and a called routine that defines:

- Obligations on the caller to create a memory state in which the called routine may start to execute.
- Obligations on the called routine to preserve the memory-state of the caller across the call.
- The rights of the called routine to alter the memory-state of its caller.

The standard also defines how an external agent can unwind the subroutine activation stack.

This standard specifies a family of *Procedure Call Standard (PCS) variants*, generated by a cross product of user-choices that reflect alternative priorities among:

- Code size.
- Performance.
- Functionality (for example, ease of debugging, run-time checking, support for shared libraries).

Many of the variants generated are not compatible with one another because the choices on which they are based are mutually exclusive or incompatible.

This standard is presented in four sections that specify:

- A machine-level base standard.
- A set of machine-level variants.
- Constraints on the layout of activation records and function entry sequences to support stack unwinding.
- The representation of externally visible C-language—and C++ extern "C" {...}—entities.

This specification *does not* standardize the representation of externally visible C++-language entities that are not also C language entities and it places no requirements on the representation of language entities that are not visible across external interfaces.

3 INTRODUCTION

This standard embodies the fourth major revision of the APCS and second major revision of the TPCS. It is the first revision to unify the APCS and the TPCS.

3.1 Design goals

The goals of the ATPCS are to:

- □ Support Thumb-state and ARM-state equally.
- □ Support inter-working between Thumb-state and ARM-state.
- □ Favor:
  - Small code-size.
  - Functionality appropriate to embedded applications.
  - High performance.

  And where these aims conflict significantly, to standardize variants covering different priorities among them.
- □ Clearly distinguish between mandatory requirements and implementation discretion.
- □ Support alternative floating-point architectures and instruction sets.
- □ Be binary compatible with:
  - The most commonly used variant of the previous APCS.
  - The most commonly used variant of the previous TPCS.

3.2 Conformance

This standard defines how separately compiled and separately assembled routines can work together. There is an *externally visible interface* between such routines. It is common that not all the externally visible interfaces to software are intended to be *publicly visible* or open to arbitrary use. In effect, there is a mismatch between the machine-level concept of external visibility—defined rigorously by an object code format—and a higher level, application-oriented concept of external visibility—which is system-specific or application-specific.

Conformance to this standard requires:

- □ Conformance to the caller-callee contract at all publicly visible interfaces.
- □ Ubiquitous conformance to its rules of stack usage.

3.3 Processes, the memory model, and the stack

This standard applies to a single *thread of execution* or *process* (the terms will be used interchangeably). A thread of execution, or process, has a memory state defined by the contents of the underlying machine registers and the contents of the memory it can address. Memory addresses are unsigned integers.

The memory a process can address and access, without causing a run-time fault, may vary during the execution of the process.

The process can read and write the stack memory. Stack memory may be statically allocated (before run time) or dynamically allocated (at run time).

3.3.1 The stack

The stack is a *full descending* stack, meaning that the stack pointer addresses the last value pushed and the stack grows from high address to low address. At any instant of execution, the stack pointer lies within a contiguous region of memory from the *stack limit* to the *stack base*. The stack need not occupy the same region of address space at all instants and the values *stack limit* and *stack base* are not necessarily available to the process itself.

When a publicly visible function is called, the stack pointer value is 8-byte aligned.

In mathematical notation:

- □ stack limit <= stack pointer <= stack base
- □ stack-pointer = 0 modulo 8 (at publicly visible calls)
- □ The stack is the half-open interval of the process address space [stack limit, stack base).
- □ The used stack is the half open interval [stack pointer, stack base).
- □ The unused stack is the half open interval [stack limit, stack pointer).

A process does not unexpectedly write to the read-write memory or *used stack* of any other process, but the *unused stack* of a process may be unexpectedly written to.

Stated differently, a process *P1* does not interfere with the memory-state of another process *P2* unless *P2* chooses to co-operate with *P1*, but an interrupt handler may execute on the stack of the process it interrupts*.*

Note This standard cannot require that an interrupt handler *can* execute on the stack of a process it interrupts—merely that if it can and does, code conforming to this standard will be oblivious to it.

3.4 Pre-conditions and post-conditions

This standard is written in terms of *pre-conditions*—what is required or guaranteed to be true before some event—and *post-conditions*—what is guaranteed or required to be true after the event.

Because this standard defines a contract between a calling routine and a called routine, conditions are symmetrical between requirement and guarantee—a requirement on the calling routine provides a guarantee to the called routine and conversely.

Conditions are written in terms of the values of components of the memory-state such as machine registers. VAL(r0) denotes the value of r0. The notation: VAL{r0, r1, ...} means the set of values {VAL(r0), VAL(r1), ...}. PRE(r0) denotes the value of r0 before an event of interest. The notation: PRE{r0, r1, ...} means the set of values {PRE(r0), PRE(r1), ...}. An indexed range is denoted by ‘-’. For example, {r0-r3} means {r0, r1, r2, r3}.

An example of pre- and post-conditions is:

> **Pre**: *stack limit* <= VAL(SP*)* <= *stack base*, VAL(LR) = *return address*
> {execute *routine*}
> **Post**: VAL{r4-r11, SP} = PRE{r4-r11, SP}, VAL(PC) = PRE(LR)

(The call of *routine* preserves r4-r11 and SP, and the PC value on return is that given in LR before the call).

4 THE BASE STANDARD

This base standard defines a machine-level, integer-only calling standard common to ARM and Thumb and a machine-level floating-point standard for ARM. The Thumb instruction set does not include coprocessor instructions, there is no machine-level floating-point standard for Thumb.

4.1 Machine registers

There are 16, 32-bit integer registers visible to the ARM and Thumb instruction sets. These are labeled r0-r15 or R0-R15. This specification uses the lower case name when there is no special role for the register.

Register	Synonym	Special	Role in the procedure call standard
r15		**PC**	The Program Counter.
r14		**LR**	The Link Register.
r13		**SP**	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8	FP	ARM-state variable-register 8. ARM-state frame pointer.
r10	v7	**SL**	ARM-state variable-register 7. Stack Limit pointer in stack-checked variants.
r9	v6	**SB**	ARM-state v-register 6. Static Base in PID,/re-entrant/shared-library variants
r8	v5		ARM-state variable-register 5.
r7	**v4**	**WR**	Variable register (v-register) 4. Thumb-state Work Register.
r6	**v3**		Variable register (v-register) 3.
r5	**v2**		Variable register (v-register) 2.
r4	**v1**		Variable register (v-register) 1.
r3	**a4**		Argument/result/scratch register 4.
r2	**a3**		Argument/result/ scratch register 3.
r1	**a2**		Argument/result/ scratch register 2.
r0	**a1**		Argument/result/ scratch register 1.

The first four registers r0-r3 are used to pass parameter values into a routine and result values out of a routine, and to hold intermediate values within a routine (but, in general, only *between* subroutine calls). In ARM-state, register r12—also called IP—can also be used to hold intermediate values *between* subroutine calls.

Typically, the registers from r4 to r11 are used to hold the values of a routine’s local variables. They are also labeled v1-v8. Only v1-v4 can be used uniformly by the whole Thumb instruction set (shown emboldened).

In all variants of the procedure call standard, registers r12-r15 have special roles. In these roles they are labeled IP, SP, LR and PC (or ip, sp, lr, and pc, but this specification uses the upper case name for the special role).

In some variants of the procedure call standard, r9 and r10 also have a special role. In these roles, r9 is labeled SB and r10 is labeled SL (or sb and sl).

Only registers r0-r7, SP, LR and PC are ubiquitously available in Thumb state. Their synonyms and special names are shown **emboldened**. Few Thumb instructions can access the *high registers,* v5-v8, SB, SL and IP.

In Thumb-state, r7 is often used as a work register and is also labeled WR.

4.2 Floating point registers

This standard supports two different floating-point architectures and instruction sets:

- The FPA architecture.
- The VFP architecture.

The FPA architecture has eight floating-point registers, f0-f7, each of which may hold a value of single, double or extended precision. We name these registers s0-s7, d0-d7 and e0-e7, according to their precision, noting that s0, d0, e0, and f0 designate the same register.

The VFP architecture has sixteen double-precision registers, d0-d15, each of which may instead be used as two single-precision registers, s0-s31. There is no extended precision and d5, for example, overlaps s10 and s11.

Code for the FPA (VFP) architecture cannot execute on a VFP (FPA) implementation.

The VFP architecture has two modes of operation:

- Scalar mode (vectors of length 1).
- Vector mode (vectors of length > 1).

This standard applies only to scalar mode operation. Use of vector mode is outside of the scope of this standard. On calling to, or returning from, a publicly visible function, a VFP floating point unit must be in scalar mode.

4.3 Subroutine call

Both the ARM and Thumb instruction sets contain a primitive subroutine call instruction, *branch with link*, or BL.

The effect of executing BL is to transfer the sequentially next value of the program counter—the *return address*— into the link register (LR) and the destination address into the program counter. The result is to transfer control to the destination address, passing the return address in LR as an additional parameter to the called routine.

Control is returned to the instruction following the BL when the called routine copies its initial value of LR to PC (see also section 5.1, *Inter-working between ARM-state and Thumb-state*).

Routine call can be implemented by any instruction sequence that has the effect:

LR ← *return address*
PC ← *destination address*
...
return address:
...

For example, in ARM-state, to call the routine addressed by, say, r4 do:

```
MOV    LR, PC       ; VAL(PC) = . + 8
MOV    PC, r4
...                 ; return to here after the call
```

In the base standard, a subroutine call preserves the values of r4-r11 and SP.

Note In the limit-checked variants of this standard, SL (r10) is neither preserved nor altered by the called routine itself, but only by limit-checking support code (that is not ATPCS compliant).

Note Return is usually to the instruction following the call sequence, but this standard does not require that.

A called routine need not preserve the values of r0-r3, IP (r12) and LR (r14). Formally:

> **Pre**: *stack limit* <= VAL(SP) <= *stack base*, VAL(SP) = 0 Modulo 8, VAL(LR) = *return address*
>
> {*called routine*}
>
> **Post**: VAL{r4-r11, SP} = PRE{r4-r11, SP}, VAL(PC) = PRE(LR) (**if not limit-checked**)
> **Post**: VAL{r4-r9, r11, SP} = PRE{r4-r9, r11, SP}, VAL(PC) = PRE(LR) (**if limit-checked**)

Here, {*called routine*} denotes the execution of instructions dynamically between the BL instruction that calls the routine and the instruction immediately following the BL. That is, the pre- and post-conditions apply across all instructions executed *after* the call sequence but before the instruction returned to.

Note This definition permits a fixed tree of non-public calls rooted at a publicly visible interface to be treated as a single call for ATPCS conformance purposes.

4.4 Parameter passing

Parameter passing is defined using a two level conceptual model:

□ Each source language parameter value is (conceptually) converted to a machine parameter value. The conversion depends on whether there is a floating point unit. The identity of a source language parameter is preserved by the (conceptual) conversion.

□ Machine parameters are marshaled into parameter lists.

There are two parameter-passing schemes according to whether or not the called routine accepts a variable number of arguments or a fixed number of arguments.

A machine-level parameter value is a value of one of the fundamental machine types— the types of the underlying machine registers—or a vector thereof. Specifically:

□ One or more 32-bit integers.

□ One to four single-precision floating-point numbers.

□ One to four double-precision floating-point numbers.

Note This standard does not regulate the passing of values of extended-precision floating-point type.

A source language parameter value is converted to a machine parameter value as follows.

□ An integer value narrower than 32 bits is widened to 32 bits in a way that preserves both the sign and the range of the value.

□ A 64-bit integer value is treated as two 32-bit integer values.

□ A homogeneous floating-point value (such as **double**, **double complex**, or a structure containing 3 **double** fields) is converted to:

  - A floating-point value of the corresponding machine type (here a sequence of 1, 2, or 3 double precision values) if the called routine will use floating-point hardware.
  - A sequence of integer machine words, *as if* by copying the value into consecutive memory words, if the called routine will simulate floating-point operations in software.

□ A value of any other type (such as a structured value) is converted to a sequence of 32-bit integer words, *as if* by copying it into consecutive memory words.

Section 7, *ARM C and C++* Conventions, describes in more detail how C and C++ source language entities map to the machine level.

4.4.1 Variable number of parameters (variadic routines)

The list of machine parameter values is converted to integer machine words *as if* by storing each value in turn into consecutive memory words. These integer parameter words are passed to a variadic routine *as if* by:

- □ Loading the first 4 words into integer registers a1-a4 (lowest addressed into a1). If there are fewer than 4 words, they are loaded into a1-a3, a1-a2, or a1 only.
- □ Pushing remaining words onto the stack in reverse order (so the first remaining parameter word is at VAL(SP), the second at VAL(SP)+4, and so on).

Note Consequently, a floating-point value can be passed in integer registers, or can be split between an integer register and memory.

4.4.2 Fixed number of parameters

Machine-level parameter values are passed to a non-variadic routine *as if* marshaled by the following algorithm:

- □ Each floating-point value is examined in turn. If it fits into a contiguous sub-range of the remaining floating-point argument registers of the appropriate sort, it is allocated to the lowest numbered such sub-range and removed from the argument list. The details of the assignment depend on the selected floating-point architecture (see section 4.6, *The FPA procedure call standard* and 4.7, *The VFP (scalar mode) procedure* call standard).
- □ While a tail of a1-a4 is unoccupied and at least one integer parameter value remains:
  - The next integer argument is moved to the head of the argument list.
  - As much as possible of that value is allocated to the remaining tail of a1-a4. Any residuum is left at the head of the argument list.
- □ Remaining parameter values are pushed onto the stack in reverse order.

Note A machine-level floating-point value is passed entirely in floating-point registers or entirely in memory.

Note At most one parameter value can be split between integer registers and the stack.

4.5 Result return

A procedure returns no result.

A function returns a single value of integer sort or floating-point sort. A value occupies 1 or more words.

A function of integer sort:

- □ Must return a 1-word value in a1.
- □ May return a value of length 2-4 words in a1-a2, a1-a3, or a1-a4, respectively.
- □ Must return a longer value indirectly, in memory, via an additional address parameter.

A function of floating-point sort:

- □ Must return a scalar floating-point value in f0 or d0, as appropriate to its precision.
- □ May return a vector floating-point value in f0-fN, or d0-dN, where N depends on the selected floating-point architecture (see section 4.6, *The FPA procedure call standard* and 4.7, *The VFP (scalar mode) procedure* call standard).

Section 7, *ARM C and C++* Conventions, describes how source language entities of various lengths are returned.

4.6 The FPA procedure call standard

The first four registers, f0-f3, can be used:

- To pass floating point values into a routine.
- To pass floating point results out of a routine.
- As scratch registers within a routine.

Registers f4-f7 are preserved across a routine call. Formally:

{*called routine*}

Post: VAL{f4-f7} = PRE{f4-f7}

Saving and restoring preserved registers

The contents of f4-f7 can be saved using a single SFM instruction and restored using a single LFM. Each value saved or restored occupies three words (12 bytes).

Floating-point values

In the FPA architecture, single- and double-precision values conform to the IEEE 754 standard formats. The most significant (exponent-containing) word of a floating point value has the lowest memory address, independent of the byte order within words.

Note When used little endian, double-precision values are neither pure little endian nor pure big endian.

IEEE rounding modes and exception enable flags

The ATPCS neither constrains on entry to a conforming routine, nor guarantees on exit from a conforming routine:

□ The IEEE rounding mode.

□ The IEEE exception enabling state.

4.7 The VFP (scalar mode) procedure call standard

Registers s0-s31 overlap registers d0-d15, with d0 using the same resources as s0-s1, d1 the same resources as s2-s3, and so on.

The first eight double precision registers, d0-d7 (single precision registers s0-s15), may be used:

- To pass floating point values into a routine.
- To pass floating point results out of a routine.
- As scratch registers within a routine.

Registers d8-d15, and s16-s31 are preserved across a routine call. Formally:

{*called routine*}

Post: VAL{d8-d15} = PRE{d8-d15}, VAL{s16-s31} = PRE{s16-s31}

Floating-point argument values are assigned to floating-point registers by assigning each value in turn to the next free, contiguous sub-range of register of the appropriate type. For example, in passing:

1.0 (double) 2.0 (double) 3.0 (single) 4.0 (double) {5.0, 6.0} (single complex) 7.0 (single)

the assignment of parameter values to registers looks like:

Double view	d0		d1		d2		d3		d4		d5
Single view	s0	s1	S2	s3	s4	s5	s6	s7	s8	s9	s10
Argument view	1.0		2.0		3.0	7.0	4.0		5.0	6.0	

Saving and restoring preserved registers

The contents of the upper-half register bank can be saved/restored as bit patterns, without interpretation as single- or double-precision numbers, using a single FSTMX/FLDMX instruction. N+1 words are transferred when N single-precision registers are saved/restored. The contents of the words transferred are unspecified.

Floating-point values

In the VFP architecture, single- and double-precision values conform to the IEEE 754 standard formats. Double-precision values are treated a true 64-bit values:

- When used little endian, the more significant (exponent containing) word of a two-word double value has the higher address.
- When used big endian, the more significant word has the lower address

Note When used little endian, the order of words within a double precision value is the opposite of that for an FPA double-precision value.

Constraints on vector mode

On entry to and exit from a publicly visible routine conforming to the ATPCS:

- The vector length is 1.
- The vector stride is 1.

IEEE rounding modes and exception enable flags

The ATPCS neither constrains on entry to a conforming routine, nor guarantees on exit from a conforming routine:

- The IEEE rounding mode.
- The IEEE exception enabling state.

4.8 The no floating-point hardware procedure call standard

Each floating-point argument value is converted to integer words *as if* by storing the value to memory.

The resulting integer values are passed as described in section 4.4, *Parameter passing*.

A scalar floating-point result is returned as 1 or 2 integer values (in a1 and a2) depending on its precision.

When a double-precision value is passed in two consecutive integer registers, the lower numbered register contains the word corresponding to the lower addressed word of the value's representation in memory.

Note Little endian, the order of words is reversed between the FPA and VFP representations.

5 THE STANDARD VARIANTS

5.1 Inter-working between ARM-state and Thumb-state

Code is built for inter-working if:

- Each subroutine changes instruction set to that of its caller on returning to its caller.
- Each call through a subroutine pointer exchanges instruction set to that of the called routine.
- Each subroutine pointer encodes the instruction set of its target in its least-significant bit (1 => Thumb, 0 => ARM).

Note In general, you do not need to be concerned with the instruction set of a routine you call directly. The tool chain is expected to insert an inter-working call veneer or patch the call site at link time.

Use the *Branch and Exchange (BX)* instruction to branch to a destination and simultaneously change instruction set to that specified by the least-significant bit of the destination address.

An inter-working return can be performed using:

```
      BX          LR
```

In ARM architecture version 5T, a load (but not a move) to the PC also restores the instruction-set state, allowing an inter-working return to be performed using LDR, LDM, or POP.

5.2 Read-only position-independence—PIC

When a *read-only (RO)* segment of a program can be loaded and used at any address, the program segment is *read-only position-independent (ROPI)*. A program is ROPI if all of its RO segments are ROPI.

Often, a program has only one read-only segment containing all of its code, and not much else, so ROPI is loosely called *position-independent code (PIC)*. Nonetheless, this standard is written in terms of ROPI.

The practical consequences of ROPI are:

- Every reference from an ROPI segment to the same ROPI segment must be a PC-relative reference (because this standard defines no other base register for an RO segment).
- Every reference from an ROPI segment to a different ROPI segment must be a PC-relative reference to a segment that is repositioned in lock step with the first.
- The address of an RO entity cannot be stored in an ROPI segment.
- Every RW word that addresses an ROPI segment must be marked as requiring relocation whenever the ROPI segment is repositioned (by being so described in the object file or image file containing it).
- Every other reference from an ROPI segment is a reference to an absolute address or an SB-relative reference to writable data.

Note If you can determine that a RW address of an RO entity is not externally visible, you may be able to replace all uses of it by equivalent uses of read-only, PC-relative offsets.

Note If an RW section is not position-independent, an ROPI section may address it directly (the values of such addresses being fixed when the image is linked).

5.3 Read-write position-independence—PID

When a *read-write (RW)* segment of a program can be loaded and executed at any address, the program segment is *read-write position-independent (RWPI).* A program is RWPI if all of its RW segments are RWPI.

Often, a program has only one read-write segment containing writable data, and not much else, so RWPI is loosely called *position-independent data (PID)*. Nonetheless, this standard is written in terms of RWPI.

A read-write segment is position-independent only until execution commences. Thereafter, it cannot be repositioned until the program restores the initial conditions that hold prior to execution.

5.3.1 Position-independent data addressing

An RWPI segment can be re-positioned until it is first used so an RO segment must not refer to it directly by address. Instead, an address is calculated by adding a read-only offset, fixed at link time, to a static base register whose value is fixed at run time.

In RWPI variants of the procedure call standard, a read-only address constant is represented by an offset from SB. By convention, SB addresses the first byte of the lowest addressed RWPI segment of the program.

Note In this addressing model there is only one SB so there can only be one independently positioned RWPI program segment if RO references to RWPI segments really are read-only and cannot be relocated. All RWPI segments must be displaced identically unless the RO references to them are writable.

5.3.2 RWPI defined

A read-write segment is RWPI if:

□ Every reference from an RO segment to the RWPI segment is constructed by adding an invariant (read-only) offset to the value of SB (SB addresses the base of the RWPI segments).

□ Every RW word that initially addresses an RWPI segment is marked as requiring relocation whenever the RWPI segment is repositioned (by being so described in the object file or image file containing it). This allows a segment to be repositioned whenever the initial conditions are restored.

Note If you can determine that an RW address of an RWPI entity is not externally visible, you may be able to replace all uses of it by equivalent uses of SB-relative offsets.

Note Read-write code should use PC-relative addressing wherever possible but is permitted to use RW address-constants to address an RW segment (and *must* do so to address an ROPI segment).

Note If an RO segment is not position-independent, an RWPI segment may address it directly and need not note such values for relocation (these values being fixed when the image is linked).

In RWPI variants of the ATPCS, the following additional pre-condition holds before every routine call:

> **Pre**: VAL(SB) = *base address of static data segments*
>
> {*called routine*}

The post-condition VAL(SB) = PRE(SB) is guaranteed by the base standard.

This pre-condition need not hold everywhere—merely on entry to each publicly visible routine. Consequently, an ARM routine that neither calls publicly visible routines nor uses SB is permitted to use v6 as a variable register.

5.4 Re-entrant code

A program which is RWPI is also *re-entrant*, meaning that several processes may thread it concurrently without interfering with one another, each process having its own copy of the program’s read-write segments addressed by its own value of SB.

5.5 Shared libraries

A shared library is a re-entrant program to which other programs may link at execution time. In general, both the library and its client have read-write (RW) segments and the process of linking to a shared library creates a copy of the library's RW segment for its client.

Shared libraries are statically linked independently of one another and independently of their clients. As a result, different values of SB are needed to address the separately linked RW segments in a program that uses one or more shared libraries. A new value of SB—or a local SB value—must be established when control passes from:

- The code of the client to the code of a library.
- The code of a library to the code of a different library

On returning, the value of SB must be as it was prior to the call.

In the *shared-library* variant of the ATPCS, a new SB value is established after entry to a shared-library routine that needs to use SB. The new value need not be established until it is needed and a routine that makes no use of SB need not establish a new value. Formally:

Pre: VAL(SB) = *address of any static data segment*

{*called routine*}

The post-condition VAL(SB) = PRE(SB) is guaranteed by the base standard.

5.6 The shared-library data-addressing architecture

The shared-library data addressing architecture extends the RWPI data addressing architecture (see section 5.3.1, *Position-independent data addressing*):

- Each shared library has a unique identity represented by a small integer (the library index) bound into the library code. A library index may be relocated when a library is loaded into RAM, or it may be bound statically. A library's index is the same in every process threading it.
- Each (per-process) static data segment begins with a table of four pointers into a process data table. They point to entries 0, 32, 64, and 96 of the process data table, respectively.
- The Nth entry in a process data table points to the start of the static data segment for library N.

In ARM-state, a routine can locate its static data using the code sequence:

```
LDR  SB, [SB, #0]
LDR  SB, [SB, #my_index]       ; my_index may be relocated
```

The base standard guarantees that SB is restored on exit from the subroutine.

In Thumb-state, SB is a high register that cannot be used directly so a subroutine can locate its static data using:

```
MOV  LSB, SB                   ; LSB any low register
LDR  LSB, [LSB, #my_segment]   ; 0, 1, 2, or 3 ...
LDR  LSB, [LSB, #my_index      ; ... and my_index may be relocated
```

A Thumb-state subroutine does not alter SB so it does not need to restore it.

Note These code sequences do not need to be inline in their using routines. Each can be replaced by a call to a special (non-ATPCS-conforming) leaf routine, saving some space and reducing the number of locations dependent on the library index, but costing some execution time.

Note In ARM-state, most non-leaf routines and every static-data-using leaf routine bears the cost of SB's fixed role by losing a v-register (but non-static-data-using leaf routines may use v6). In Thumb-state, the local static base (LSB) is only needed in static-data-using routines where common sub-expression elimination and register allocation can be applied to it together with user variables.

5.7 Stack limit checking

In the stack-limit-checked variants of the ATPCS:

- SL points at least 256 bytes above the stack limit described in section 3.3,*Processes, the memory model, and the stack*.
- SL is neither preserved nor altered by code that complies with these variants of the ATPCS (it is altered only by run-time support code).
- VAL(SP) >= stack limit, at all instants of execution.

Note The limit-checking code must also run on its caller's stack, so in general, SP must be set more than 256 bytes above *stack limit*—perhaps considerably more, but this is a run-time support issue.

Note If an interrupt handler can run on the stack, SL must be set even further above *stack limit.*

Formally:

Pre: VAL(SP*) <= stack base*, VAL(SP) >= VAL(SL) >= *stack limit* + 256, VAL(LR) = *return address*

{*called routine*}

Post: VAL{r4-r9, r11, SP} = PRE{r4-r9, r11, SP},
VAL(SP*) <= stack base*, VAL(SP) >= VAL(SL) >= *stack limit* + 256, VAL(PC) = PRE(LR)

Note *Stack base,* and *stack limit* are not necessarily preserved across the call.

SL >= *stack limit* + 256, and VAL(SP) >= *stack limit* let a called routine drop SP up to 256 bytes below SL before calling limit-checking code.

If a routine uses less than 256 bytes of stack:

□ If it is a leaf routine, it need not check the stack limit at all.

□ If it is a non-leaf routine, it may use a very simple limit-checking sequence like that shown below.

```
ARM                           Thumb

SUB  SP, SP, #size            ADD  SP, #-size
CMP  SP, SL                   CMP  SP, SL
BLLO __ARM_stack_overflow     BLO  __Thumb_stack_overflow
```

Note For this purpose, a *leaf routine* is one which calls no other routine or in which every call is a tail continuation (effectively, a call made from this routine's caller).

Checking for overflow in a routine that uses more than 256 bytes of stack space, is more complicated. The routine cannot simply subtract the frame size from SP without risking violating the global invariant VAL(SP) >= *stack limit*. In this case, a new value of SP must be *proposed* to the limit-checking code using a sequence like:

```
ARM                           Thumb

SUB  IP, SP, size             LDR  WR, NegativeSize
CMP  IP, SL                   ADD  WR, SP
BLLO __ARM_stack_overflow     CMP  WR, SL
                              BLO  __Thumb_stack_overflow
```

Note The names `__ARM_stack_overflow`, `__Thumb_stack_overflow` are illustrative and do not reflect or standardize any actual implementation.

5.8 Chunked stacks

A chunked stack is a non-contiguous, linked list of contiguous chunks with the following properties:

- You can extend a chunked stack by allocating an additional chunk anywhere in memory. There are no constraints on the ordering of chunks in the address space
- Within a chunk, activation records are allocated in descending address order. At all observable instants of execution, SP points to the lowest used address of the most recently allocated activation record.
- At all observable instants of execution, SL identifies the same chunk as SP points into. SL points at least 256 bytes above the lowest usable address in the chunk.
- A chunked stack must be limit-checked. It can be extended only when a limit check fails.

Corollaries

The requirement that, at all observable instants of execution, SP and SL point into the same chunk means that on changing stack chunks either:

- SP and SL must be loaded atomically.
- Or, an interrupt handler cannot run on the stack of the process it interrupts.

In ARM-state, SP and SL can be loaded simultaneously using:

```
LDM ..., {..., SL, SP}
```

In general, this means that return from a routine executing on an extension chunk to one executing on an earlier-allocated chunk should be through an intermediate routine activation, specially fabricated when the stack was extended.

6 STACK UNWINDING

6.1.1 Background

Stack back-tracing code, chunked stack extension code, C++ *exception* handlers, and debuggers all need to *unwind* a stack. That is, they need to access the register-state from each activation record in a chain of subroutine activations, working up the stack from a called routine through its calling routine, and so on.

There are several approaches to making the state of the stack intelligible to an external agent:

- How to unwind a stack frame may be described in tables used by the agent.
  - The ARM SDT supports DWARF2.0's target-independent frame unwinding descriptions for debuggers. This way, a hosted unwinding agent imposes neither table overhead, nor layout restrictions, on a target.
  - ARM C++ uses a compressed description to drive unwinding by its exception handlers.
- The layout of a stack frame may be prescribed (partly or wholly) and frames may be chained together directly through a real, or virtual, frame pointer, as in some variants of earlier versions of the APCS.
- An unwinding agent may unwind a fixed stack frame by interpreting a routine's entry sequence, undoing stack adjustments in reverse order. An auxiliary table is used to locate an entry point from a given PC value.
- An unwinding agent may unwind an activation record by interpreting a routine exit sequence. Again, an auxiliary table is needed to locate the appropriate exit sequence. In some circumstances an exit sequence may be directly executed after patching the routine's return address (but more table support is needed).

Requiring the use of a frame pointer everywhere ties up a register. This was specified by early variants of the APCS, but it proved unacceptable to customers. The earlier TPCS never featured a frame pointer.

Using a virtual frame pointer requires an additional table, indexed by PC value, which gives the offset of the virtual frame pointer from the stack pointer. Requiring a virtual frame pointer increases the size of a program.

6.1.2 What this standard defines

This standard defines:

□ Alternative activation record structures.

□ Matching constraints on entry and exit sequences.

Each activation record in the stack must conform to one of the alternative structures.

When stack unwinding is not required there is no overhead from auxiliary tables. When unwinding is required, the auxiliary tables can be space efficient.

6.2 Allowed alternatives for unwinding

A routine must use:

□ A fixed-size activation record (obey the stack-moves-once condition).

□ Or, a frame pointer.

The following subsections describe:

□ How to unwind an activation record.

□ Restrictions on code generation and register usage needed to support unwinding.

6.2.1 Basic routine shape

The basic shape of a routine is:

Entry sequence
Routine body
Exit sequence

The entry sequence begins at the instruction addressed by the routine's machine-level address.

A routine can have multiple entry points. You can think of each additional entry sequence as a tail-continued routine in its own right.

A routine can have multiple exit sequences. Usually, it is only profitable to replicate a 1-instruction exit sequence.

6.2.2 The stack-moves-once condition

Throughout the body of a routine that obeys the *stack-moves-once* condition, the value of SP must be identical to the value of SP at the end of the entry sequence.

Strictly, this condition only needs to hold at points where control might pass—whether via a chain of calls or via a run-time trap—to an unwinding agent.

This standard allows any complying routine to pass control to an unwinding agent so, in general, the condition must hold at all calls to routines that are publicly visible or call publicly visible routines.

In practice, we often use the stronger form of the condition in which code in a routine body does not alter SP.

Corollaries

You may observe that the routine entry code and exit code:

- Must not trap or raise exceptions.
- Must not call routines that might call an unwinding agent (if a routine is called, it will have to comply with extra restrictions not required by this standard).

Failure to comply with these conditions may cause stack unwinding to fail.

6.2.3 Unwinding a fixed size activation record by interpreting an entry sequence

You can unwind an activation record created by a routine that obeys the stack-moves-once condition by interpreting the routine's entry sequence to:

- Undo the effect of SP-decrementing instructions.
- Restore the callee-saved register-state.

With the exception of stack-limit checks, there are no PC-modifying instructions in an entry sequence, so there is no need to interpret beyond the first-found—possibly conditional—branch, or other PC-modifying instruction. Because a routine ends with a PC-modifying instruction (a return), you cannot interpret too far using this rule. Similarly, there can be no SP-incrementing instructions in an entry sequence.

The entry sequence of a stack-limit-checked routine with a small activation record (<= 256 bytes) ends with adjusting SP, comparing SP with SL, and branching conditionally to, or around, the stack overflow-handling code.

The entry sequence of a stack-limit-checked routine with a large activation record (> 256 bytes) ends with proposing a new value of SP, comparing this register with SL, and branching conditionally to, or around, the stack overflow-handling code.

The register holding the new value of SP (possibly SP itself) can be deduced from the `CMP` instruction.

Summary of constraints

- □ A stack-limit-checked entry sequence ends with:
  - An inline comparison of SP, or proposed-new-SP, with SL.
  - Instructions to call the overflow handler and adjust SP.
- □ Otherwise, an entry sequence contains neither PC-modifying, nor SP-incrementing, instructions.
- □ All SP-decrementing instructions occurring before the first (possibly conditional) PC-modifying instruction of a routine body belong to the routine entry sequence.

The last constraint allows the weaker (more precise) definition of stack-moves-once to be used (following any instruction that might modify the PC) without hazard to stack unwinding.

6.2.4 Unwinding a fixed size activation record by executing an exit sequence

If the unwinding agent can:

- □ Find the start of the exit sequence appropriate to the current PC value.
- □ Locate the return address (which must be in LR or at a fixed offset from SP known to the agent).

It can patch the return address to an address in the unwinding code and branch to the exit sequence. The exit sequence will do precisely what is needed to unwind the activation record, restore the callee-saved register-state, and return to the unwinding code.

Summary of constraints

- □ For every PC value in the routine body from which stack unwinding might be provoked, there must be a statically determined (data-independent) exit point.
- □ At each exit point:
  - The size of the activation record must be fixed.
  - Or, the return address must be in LR.

6.2.5 Constraints on frame pointers

This standard neither requires, nor precludes, the use of a frame-pointer register. Any such use is local to the activation of a routine built this way.

If a frame-pointer is used, it must be a callee-saved register.

- □ In ARM-state, the frame-pointer register is r11 (FP).
- □ In `Thumb`-state, any of r4-r7 can be used.

The frame pointer points immediately above the saved return address.

- □ In a non-variadic routine, this is the value of SP on entry to the routine.
- □ In a variadic routine, the frame pointer points to the list of stacked arguments—it is the value of SP after pushing the argument registers.

A frame pointer holds the same value at all points within the routine body from which stack unwinding might be provoked. Effectively, a frame pointer holds the same value throughout a routine.

6.2.6 Unwinding an activation record using a frame pointer

If an unwinding agent can determine from its supporting tables that an activation record belongs to a frame-pointer-using routine, it can unwind the activation record by:

- Interpreting the entry sequence to restore the callee-saved register-state.
- Setting SP to the value of the frame pointer.
- Incrementing SP by 4 times the number of argument registers saved if the routine is variadic.

Or, by intercepting the return address (in LR or at –4[FP]) and executing an exit sequence, as described in section 6.2.4, *Unwinding a fixed size activation record by executing an exit sequence*.

The supporting tables must encode:

- The register used for the frame pointer.
- The number of argument registers saved by the routine if it is variadic.
- The location of the return address (LR or –4[FP]).

6.3 The shape of routine entry

The shape of routine entry is determined by the set of procedure call standard variants and some choices about the order in which tasks are performed in an entry sequence.

A routine entry sequence performs some or all of the following five steps in order:

1 Push some, or all, of the argument registers {a1-a4} to make a contiguous, stacked argument list.

2 Push integer registers that must be preserved and may be written to by this routine
(a leaf function that corrupts only a1-a4 and IP need save no integer registers).

3 Establish a frame pointer or argument pointer.

4 Push floating-point registers that must be preserved and may be written to by this routine or its called routines
(a leaf function that corrupts only floating-point argument registers need save no floating-point registers).

5 Reserve any further stack space the routine needs by performing one of:

   a) Decrement SP unconditionally (unchecked, contiguous stacks only).

   b) Decrement SP by less than 256 and perform the small-frame limit check (small frames only).

   c) Propose a new value of SP in IP (ARM) or WR (Thumb) and perform the big-frame limit check.

Notes

- Establishing a new value of SB does not affect the stack in any way and is not part of the entry sequence.
- Because the frame-pointer is a callee-saved register, a frame-pointer cannot be established until after some registers have been saved (after step 2).
- There may be a sequence of push-integer-registers instructions and a sequence of push-floating-point-registers if the length of multiple transfers has been limited to improve interrupt latency.
- There may in any case be a sequence of push-floating-point-registers, depending on the floating-point architecture. For example, VFP could require one instruction to save preserved registers; one to save address-taken double-precision arguments; and one to save address-taken single-precision arguments.
- There are many alternative code sequences to decrement SP or propose a new value of SP.
- Instruction scheduling may reorder instructions and, in the absence of control transfers, reorder routine entry instructions with instructions from the immediately following routine body.

7 ARM C AND C++ CONVENTIONS

This section describes how ARM compilers map C and C++ language features onto the machine-level standard described in sections 4 to 6.

7.1 ANSI C and C++ argument passing conventions

The way argument values are passed at the machine level depends on the choice of target floating-point architecture. Machine-level argument passing is described in:

- Section 4.4, Parameter passing.
- Section 4.6, *The FPA procedure call standard* and its subsection *IEEE rounding modes and exception enable flags*.
- Section 4.7, *The VFP (scalar mode) procedure call standard* and its subsection *IEEE rounding modes and exception enable flags*.

The ATPCS neither constrains on entry to a conforming routine, nor guarantees on exit from a conforming routine:

- The IEEE rounding mode.
- The IEEE exception enabling state.

This section describes in more detail how C and C++ source language entities are converted to sequences of machine-level values by ARM compilers.

Note The Thumb instruction-set cannot access coprocessors directly, so in Thumb-state you must use the *no floating-point hardware* variant.

Conversion of narrow argument values

A narrow integer argument (type **char**, **short**, **enum**, and so on) is widened to fill a 32-bit word by zero-extending it or sign-extending it as appropriate to its type.

Conversion of long integer (long long, __int64) values

A long integer is converted to 2 argument words *as if* by storing it to memory then copying the low-address word to the first argument and high-address word to the second argument.

Conversion of floating-point values

Conceptually, primitive floating-point values (**float**, **double**) are handled at the machine level and require no conversion (see section 4.4, *Parameter passing*).

Conversion of homogeneous floating-point structured (struct) values

A source language value that when laid out in memory consists of between 1 and 4 consecutive floating-point values of the same precision (single or double), is converted to a machine-level floating-point value of the same length and sort.

Conversion of other structured (struct, union) values

A structure value is converted to argument words *as if* by storing it to memory then copying the sequence of words it overlaps in increasing address order.

If a structure does not occupy an integral number of words, the final argument word can contain 1, 2, or 3, undefined bytes. For little-endian targets, these are always the most significant bytes. For big-endian targets they are the least significant bytes.

7.2 Narrow arguments

In C++ and in ANSI-C in the presence of a function prototype, a called function can assume that a received parameter value is in the range of its parameter type. For example, if the parameter type is **unsigned char**, its value can be assumed to lie in the range UCHAR_MIN to UCHAR_MAX. The calling function must ensure that the argument value lies in the range of the called function's parameter type.

In pre-ANSI C and in ANSI-C using pre-ANSI function declaration syntax, narrow argument values are widened by the default promotion rules. A called function must defensively narrow the values of its narrow parameters because the calling function cannot ensure that the argument values are in range (when the call is made, nothing is known about the types the called function's parameters).

Analogous statements hold for the floating-point types **float** and **double**.

In C++, and in ANSI-C in the presence of a function prototype, a calling function does not convert a **float** argument value to **double** if the called function's parameter type is **float**.

If a **float** argument value matches an ellipsis ('...') in the called function's parameter list, or is being passed to a pre-ANSI C function of unknown parameter type, the calling function must convert the value to **double**.

7.3 Result return

7.3.1 Non-floating-point results

- □ An integral result no wider than a word is returned in a1.
- □ An __int64 result is returned in a1, a2 (high address part in a2).
- □ A structure or union result no larger than a word is returned in a1.

Larger structured values are returned indirectly via an additional address argument that is passed first in the argument list. For example:

```
struct S f(void);
struct S sv = f();
```

Is compiled as if you had written:

```
void f(struct S *);
struct S sv;
f(&sv);
```

7.3.2 Floating-point results

Floating-point results (no-floating-point-hardware variants)

A single-precision result is returned in a1.

A double-precision result is returned in {a1, a2} (high address part in a2).

Floating-point results (FPA variant)

A floating-point result is returned in f0, independently of its precision.

Floating-point results (VFP variant)

A single-precision result is returned in s0, a double-precision result in d0.

7.3.3 Value-in-registers result return

The ARM C/C++ compiler permits a C/C++ structure-value-returning function to be declared with the storage class __value_in_registers.

Four or fewer homogeneous floating-point values in the FPA and VFP variants

The value of a structure-valued result, that when laid out in memory consists of between 1 and 4 consecutive floating-point values of the same precision, is returned in:

- f_0-f_{n-1} in the FPA variant.

□ d_0-d_{n-1}, or s_0-s_{n-1}, depending on precision, in the VFP variant.

All other structured values

If the structure occupies *n* <= 4 words, its value is returned in a_1-a_n. Otherwise, __value_in_registers is ignored.

7.4 __shared_library

Using a datum directly exported from a shared library requires an extra indirection in how the datum is addressed.

ARM compilers use the __shared_library storage class to mark the declaration of these data.

8 RATIONALE FOR ATPCS VARIANTS

This section and its subsections are not part of the ATPCS. They are provided:

- □ As an aid to understanding the ATPCS.
- □ As guidance to 3rd-party tool developers.
- □ As documentation of the behavior of ARM development tools and libraries.

8.1 The base standard and its variants

Properties of the base standard

The base standard imposes the fewest constraints on a code generator, giving the best potential of all its variants for the smallest, or fastest, code.

The base standard's ARM-state register usage conventions are compatible with its Thumb-state conventions so the ARM-Thumb inter-working variant is supported whenever:

- □ The target has the BX instruction.
- □ BX-using code sequences are used for returning from routines and calling through function variables.
- The target conforms to ARM architecture version 5 or later, and the PC is modified by a load instruction (LDR, LDM, or POP).

The base standard's register usage conventions are compatible with the minimum functionality (most widely used) variants of the earlier APCS and TPCS, so new code conforming to the base standard has the best chance of being compatible with legacy objects and libraries.

Properties the base standard does not have

- □ Capable of inter-working between ARM-state and Thumb-state.
- □ Position-independent.
- □ Re-entrant.
- □ Stack-checked.
- □ Compatible with shared-library variants of the ATPCS.

Run-time library variants

The ATPCS variants have been designed to be orthogonal to simplify compiler support for them. This generates a large cross product of run-time library variants that potentially have to be supported.

Some of these variants are one-way compatible. That is, variant Y can be used (at greater cost) wherever variant X can be used, but not conversely. Although the cost of the functionality added by a variant may be too great to impose on a user who neither needs, nor wants, that functionality, the cost of that variant of a particular library may be acceptably low when amortized over the user's application. This can reduce the number of run-time library variants that need to be supported.

The following table summarizes the cost of some variants of the ARM C library. Section 8.5, *Derivation of library variant costs*, gives further details of how these numbers were derived. Cited percentages measure the read-only size increase relative to the size using the base ATPCS of:

- □ The subset of the ARM ANSI C Library that is written in ANSI C.
- □ Or, when easier to measure, the whole ARM ANSI C Library.

Summary of basic variants and indicative costs of use

Variant	ARM-state	Thumb-state	Recommended usage
Shared-library friendly (no use of v6)	1%	0%	Always for run-time libraries. Otherwise a user choice (can impact ARM-state performance much more).
Inter-working (returns and indirect calls) Architecture v5 or later	2% 0%	2% 0%	Always for run-time libraries when supported by the target architecture. Otherwise a user choice. Generally impacts performance by less than 2%. Always.
ROPI	0.3%	0.5%	Always? Negligible impact on space and speed.
RWPI	3.3%	1.8%	Use the shared library variant (+2.3%) for run-time libraries. Shared library is usable as RWPI but not conversely. Otherwise a user choice.
Shared library	5.6%	4.1%	Always a user choice.
Stack-limit checked	3.5%	3.9%	Always a user choice.

Measurements and extrapolations apply to the way that ARM compilers generate code and may not accurately represent what should be expected using other compilers.

8.2 ARM Shared Libraries

The ARM shared library architecture has been designed with the following in mind:

- It works the same way in ARM-state and Thumb-state and efficiently in both states.
- It implicitly supports inter-working between ARM-state and Thumb-state.
- It neither requires, nor precludes, dynamic linking, and it permits dynamic loading:
    - A system built using shared libraries can, nonetheless, be linked statically.
    - But, like a DLL, a shared library can be attached to a running process.
- It neither requires, nor precludes, a function entry veneer between a caller and a callee unless:
    - A base-standard, shared-library-friendly client must call a member of a shared library.
    - A call must swap between ARM-state and Thumb-state.
    - A call must span a larger address range than BL allows.
    - Dynamic linking is needed on first use of a library.

There are differences of detail between ARM-state and Thumb-state. The standard cannot be simultaneously:

- Rigidly symmetrical between ARM and Thumb.
- Efficient in both states.
- Derived from a base standard compatible with the earlier APCS and TPCS.

The biggest surprise of the ATPCS shared library mechanism is that the read-only part of a library embeds the library's identity in the form of a small index into a process-specific table of libraries. It is more conventional for the identity to be implicit in the stub, or procedure linkage table, used to access the library code. Unfortunately, we have been unable to make the more conventional scheme work efficiently in Thumb-state.

As a consequence, the identities of libraries committed to ROM must be allocated statically, before the ROM image is created.

The identity of a dynamically loaded library can be assigned statically or when the library is loaded. If it is assigned when the library is loaded, at least one location in the library's read-only segment must be relocated.

Whether one location or many needs to be relocated depends on how the library has been built—on whether the code establishing the new static base is inline in each static-data-using routine, or out of line in one place.

8.2.1 Basic static data addressing in a shared-library-using application

In the simplest possible program structure:

- A number of client programs and a number of shared libraries are statically linked together. Calls between functions are direct.
- The clients are built the same way as the libraries (this is not necessary—see, for example, section 8.2.3, *Base-standard clients*). By convention, the library index of the clients is 0.
- Each client executes in a separate process.
- The index of a library is the same in every process threading it.

On starting up, each client establishes the following conditions, maintained invariant by clients and libraries:

- 0[SB] points to a process data table (PDT), specific to the process.
- PDT[*N*] points to the static data for shared library *N* in this process.

On entry to any routine, SB identifies it caller's static data. After saving SB (ARM-state) or allocating a local static base (LSB) register (Thumb-state), a static-data-using routine can find its static data using LSB = 0[SB][*N*].

On return, SB is restored (ARM-state) or unchanged (Thumb-state).

8.2.2 Exported data

Although the offset of a datum within a library's static data is fixed when the library is statically linked, offsets between static data areas are not fixed, even if the entire system is statically linked. For example, the offsets between library static data areas may be different in different processes threading the libraries.

For this reason, a reference to a datum exported by another link unit (shared library or client) must be indirect, via an address relocated when the static data is allocated. In other words, access to directly exported data requires an element of dynamic linking. This standard does not specify how this done. ARM C and C++ tools use an extra indirection to address data identified as

```
    extern __shared_library exported_thing;
```

That is, the address of `exported_thing` will be loaded SB-relative, then de-referenced. The dynamic linker must relocate this address when the library is attached to the process. Details are specific to the operating environment.

8.2.3 Base-standard clients

A client program can be built non-re-entrant using:

- The shared-library-friendly (v6-avoiding) variant of the base standard in ARM-state.
- The base standard in Thumb-state.

Such code addresses its static data (usually) by loading the address of a datum PC-relative. Such code must not be threaded by more than once process.

Because such code knows how to address its static data independent of SB, it is always safe for shared library code to call it directly.

Direct calls in the other direction do not work. An SB-loading veneer must be inserted between the caller and the callee. An ARM-state procedure-linkage veneer can look like:

Exit to ARM-state only.

```
MOV       IP, PC
LDMIA     IP, {SB, PC}
DCD       data-address
DCD       function-entry
```

Exit to ARM-state or Thumb-state

```
MOV       IP, PC
LDMIA     IP, {SB, IP}
BX        IP
DCD       data-address
DCD       function-entry
```

A Thumb procedure-linkage veneer can look like:

Four or more fixed arguments.

```
PUSH      {a4}
LDR       a4, [PC, #8]
MOV       SB, a4
LDR       a4, [PC, #8]
MOV       IP, a4
POP       {a4}
BX        IP
DCD  data-address
DCD  function-entry
```

Fewer than 4 fixed arguments.

```
LDR       a4, [PC, #4]
MOV       SB, a4
LDR       a4, [PC, #4]
BX        a4
DCD       data-address
DCD       function-entry
```

The Thumb procedure-linkage veneers can exit to ARM-state at no additional cost and all veneers support a full 32-bit branch span.

8.3 Dynamically loaded libraries

The operating environment must provide operations with functionality similar to the following:

- Load, or locate if already loaded, a named library, returning a library handle. This may require a library index to be allocated and the library code to be relocated.
- Attach a library identified by a library handle to the calling process. This requires a copy of the library's static data to be allocated and initialized.
- Locate a named entry point of the library identified by a library handle, returning its address.
- Return the address of the process data table.

If the caller is a base-standard caller, the entry point must address an SB-loading procedure linkage veneer. In general, this must be allocated in the library's static data, as its data-address field must be re-located in a process-specific way. This re-location is perhaps best done on first call through the veneer. Details are specific to the operating environment.

8.4 Legacy issues

This section considers:

□ Compatibility between ATPCS and its predecessors APCS and TPCS.

□ The problems posed by old style, K&R, pre-ANSI, or *dusty deck*, C.

To scope the problem, note that:

□ The legacy issues concern:

- The compatibility of register usage conventions between ATPCS and APCS/TPCS.
- The compatibility of argument passing conventions between ATPCS and APCS/TPCS.
- The suitability of argument passing conventions for old style C.

□ The register usage conventions of the base standard are compatible with the most widely used variant of the APCS (no frame-pointer, no stack checking) and the TPCS (no stack checking).

□ The ATPCS register usage conventions for stack checking are compatible with APCS and TPCS.

□ There is neither legacy object code nor legacy assembly language for the VFP architecture, support for which is new to ATPCS.

□ Hitherto, the FPA architecture has been used relatively rarely. Software floating point (no floating-point hardware) has been the norm.

□ In the *no-floating-point-hardware* variants, the argument passing conventions are virtually unchanged from those of APCS and TPCS (but see section, 8.4.2, *Narrow arguments*, below).

□ In the *no-floating-point-hardware* variants, there are no problems with old style C (but see section, 8.4.2, *Narrow arguments*, below) because the argument passing method implicitly supports variadic functions.

This effectively scopes the legacy problem to the problems of of:

□ Passing floating point arguments in floating point registers (FPA and VFP variants).

□ Passing narrow arguments.

8.4.1 Floating point argument passing

The FPA variant of the ATPCS is compatible with the APCS *floating-point-arguments-in-floating-point-registers* variant.

There are no issues with the VFP and *no-floating-point-hardware* variants.

The key problem is that of knowing when a called function is variadic because floating-point arguments must be marshaled differently for such calls.

ANSI C with prototypes, or C++

There is no problem. A language processor can always tell when a function is variadic.

Old style C under the ANSI standard

□ The ANSI standard does not allow a function declared in the old style to be variadic. This effectively requires a prototype for all user-defined variadic functions.

□ The standard permits a language processor to recognize all standard library functions by name (pertinently, printf and friends), whether or not a prototype is in scope.

□ The standard forbids user re-definition of any standard library function.

Collectively, these edicts allow a language processor to handle old style C correctly provided:

- □ The language processor recognizes the variadic members of the C library by name.
- □ The user provides a proper prototype for every variadic function that can accept a floating-point argument.

This is not particularly burdensome for users, remembering that there is a distinct performance advantage to passing floating-point arguments in floating-point registers.

Old style C not under the ANSI standard

Some language processors provide a pre-ANSI mode of operation. For example, the ARM C compiler attempt to mimic the behavior of BSD-Unix's Portable C Compiler (PCC).

In such a mode, every function is potentially variadic. This is a disaster for performance and for compatibility with the C run-time library.

A reasonable heuristic to use in these circumstances is that if the first argument value has floating-point type, the called function is *not* variadic.

- □ This heuristic certainly works for, and restores compatibility with, all functions in the ANSI C library.
- □ The heuristic fails only if a user-defined variadic function has a first parameter of floating-point type. I believe such functions are extremely rare. The possibility of failure can be detected by warning when:
  - A parameter has its address taken in such a function (potentially much less rare, and annoying), though this warning heuristic could be sharpened by warning only if the last parameter has its address taken.
  - Such as function has a (last) parameter called va_alist (potentially precise if varargs.h has been used).
- □ The heuristic is less than optimal when a user-defined, non-variadic, essentially floating-point function has a first parameter of non-floating-point type.

Other strategies are possible.

8.4.2 Narrow arguments

ATPCS requires a widened narrow argument to be in the range of its argument type.

This is a potential problem for old style C because the caller cannot know the type of a narrow argument. It is not a violation of the ATPCS because the actual argument type is the promoted type, not a narrow type.

In ANSI C, a function cannot have an argument type narrower than the default promoted type unless it has a prototype. But, if there is a prototype, there is no problem.

A C function defined in the old style must narrow its own arguments (or take the consequences). It cannot rely on its caller to do so because there is either no prototype or the prototype declares promoted types. So there is no problem with functions defined in the old style, no matter who calls them.

In short, the problem is restricted to old style C functions calling ANSI C, or C++, functions that have narrow arguments and that are declared in the old style. But, there are no such functions in the ANSI C library so there is no real problem in the ARM SDT.

Overall, it is believed that the problem of narrow arguments is of only theoretical interest. Compilers for other architectures (for example the Transputer) have relied on caller narrowing and have never seen a problem with dusty deck C from real users.

8.5 Derivation of library variant costs

Shape of the ANSI-C subset of the ARM C Library

Measure	ARM-state	Thumb-state	Comment
#(C-derived object files)	161	161	
#(functions)	230	229	
#(functions calling functions)	132	153	Tail continuation reduces the ARM count. No tail continuation in Thumb-state.
#(address constants)	278	354	
#(data address constants)	249	283	
#(functions using data address constants)	120	120	
#(non-leaf functions using static data)	75	75	
Total code bytes	42524	31160	
Total data bytes	892	888	

Sort of function		Number of functions (ARM)	Number of functions (Thumb)
No static data	Leaf function	53	31
Static data-using	Leaf function	45	45
No static data	Non-leaf function	57	78
Static data-using	Non-leaf function	75	75
TOTAL		230	229

(So, in ARM-state, about 22 functions become leaf functions because all their calls are tail-continued).

Number of arguments to publicly visible functions	#(functions)
0, 1, or 2, arguments	119
3 arguments	21
4, or more, or a variable number of, arguments	12
TOTAL	152

Cost of inter-working between ARM-state and Thumb-state

	Code Size	inline data	Inline Strings	const data	RW data	0-Init data
armlib.32l	61892	1072	1896	1544	1200	1360
armlib_i.32l	63208	1072	1896	1544	1200	1360
armlib.16l	42248	1756	1564	1428	1216	1616
armlib_i.16l	43044	1756	1564	1428	1216	1616

The code bloat for ARM is 63208/61892 = 1.0213 = +2%. For Thumb the bloat is 43044/42248 = 1.0188 = +2%.

The impact on performance is likely to be considerably less than this.

In each case, the cost is low enough that inter-working should be the default for run-time library code whenever the target instruction-set architecture supports it:

□ Always in Thumb-state.

□ For all Thumb-aware architectures in ARM-state (architectures derived from 4T and 5T).

Compatibility with shared libraries—avoiding use of v6

Code that conforms to the base standard cannot directly call code that conforms to the shared-library variant because SB will be invalid. Calls in the opposite direction are safe providing they are serialized (code conforming to the base standard is not re-entrant).

SB can be set up in a tail-continued veneer inserted between a base-standard client call site and a shared-library destination. Unfortunately, doing this corrupts a callee-saved register making the veneer incompatible with a base-standard-conforming caller.

Base-standard-conforming code can be made compatible with shared library variants via a tail-continuation veneer if no use is made of v6 (SB). The cost of this in ARM-state is very modest (1%). In Thumb-state there is usually no cost because v6 is not directly usable in Thumb-state.

ARM-state code size using v6	Code size avoiding use of v6	Size increase
42524 bytes	42956 bytes	1.0%

Unfortunately, the impact on the performance of critical user code can far exceed 1%. Perhaps 3-5% should be expected in the worst cases, which is why this variant should not be imposed as standard.

The cost of ROPI

A read-only section cannot be position-independent if it contains the address of an ROPI section. In the ARM C Library, the cost of avoiding this is quite low. A PC-relative load of a pointer to an RO entity is replaced by:

□ A PC-relative load of the offset of the entity from the current instruction.

□ An addition of the current PC-value to the loaded offset.

The first order effect is that an extra instruction is needed to compute each address common-sub-expression. A second order effect is that address offsets—unlike addresses—cannot be shared between address computations (an offset cannot be relative to 2 different PC values).

In the written-in-C subset of the ARM C Library we can calculate the approximate cost as 1 instruction per non-data-referring address constant. (Statistics were collected from a build of the library in which most functions were compiled separately, so there was little sharing of address constants between functions).

	ARM-state	Thumb-state
#(non-data address constants)	278 – 249	354 - 283
Size increase	29 * 4	71 * 2
% size increase	0.27%	0.46%

This estimate is approximate, but shows that the cost of the ROPI variant is small enough to allow it to be adopted as the standard library variant (there are no compatibility issues with non-ROPI code).

Cost of shared library relative to RWPI

Code built RWPI is not compatible with shared library code. However, code built for shared library use *is* RWPI and so is compatible with RWPI code if the run-time environment maintains the invariant 0[0[SB]] = SB (by convention, shared library index 0 is reserved for client code).

The additional cost of building for shared library 0 is 2 instructions (8 bytes) per static-data-using function in ARM-state and 3 instructions (6 bytes) in Thumb state.

C subset of the ARM C Library	ARM-state	Thumb-state
#(static-data-using functions)	120	120
Cost per function	8 bytes	6 bytes
Approximate % size increase	2.3%	2.3%

While 2.3% is an unacceptable overhead to impose on all user code, it may well be an acceptable overhead on a run-time library that will form a small proportion of the final application.

Shared library support

In comparison with the base standard, shared library support costs approximately:

- □ In ARM-state
  - One dedicated register (SB).
  - Two instructions per static-data-using function.
  - One instruction per data address constant.
- □ In Thumb-state
  - Three instructions per static-data-using function.
  - One instruction per data address constant.

C subset of the ARM C Library	ARM-state	Thumb-state
Cost of SB	1%	0%
#(static-data-using functions) * cost	120 * 8 bytes	120 * 6 bytes
#(data address constants) * cost	249 * 4 bytes	283 * 2 bytes
Approximate % size increase	5.6%	4.1%

Stack limit checking

A stack limit check costs approximately:

- □ One dedicated register (SL) and 2 instructions (8 bytes) per non-leaf function, in ARM-state.
- □ Four instructions (8 bytes) per non-leaf function, in Thumb-state.

Checking a large frame is slightly more expensive than this, especially in Thumb-state, but large frames are rare (there are none in the ARM C library, for example).

C subset of the ARM C Library	ARM-state	Thumb-state
#(non-leaf functions) * cost	132 * 8	153 * 8
Cost of SL	1%	0%
Approximate % size increase	3.5%	3.9%

Allowing for large frames, we can estimate 4% as an indicative size increase. The impact on performance should, generally, be much less than this (because leaf functions and loops dominate performance).