# Cereon

Procedure Calling Standards

Authors:
Andrei Kapustin

Revision history

| Date | Comment |
|---|---|
| 22 November 2006 | Initial draft. |
| 3 November 2007 | Documented the NPCS, TPCS and BPCS. |

# 1 Contents

# 2 Preface

This document is a definitive guide into the procedure calling standards used by Cereon platforms.

A Procedure Calling Standard (henceforth abbreviated as PCS) is a specification of:

- How arguments are passed to procedures.
- How the said procedures are invoked.
- How results are returned from these procedures.
- What register usage rules must be obeyed.

Note that, although this document describes a number of “standard” procedure calling standards, there is nothing in the world to stop a particular toolchain vendor from inventing and implementing a proprietary PCS of their own – the only drawback of that approach would be in the fact that programs compiled with the said toolchain will not be binary-compatible with those compiled by other toolchains.

Note also that the procedure calling standard described herein have a lot in common, (including register usage rules); therefore, it may be possible, in some cases, for a procedure conforming to one calling standard to call a procedure that conforms to another calling standard (for example, a procedure that conforms to BPCS can be called by a procedure conforming to any other calling standard described); however, this is not recommended.

# 3 Common procedure calling standard

The Common Procedure Calling Standard (abbreviated henceforth as CPCS) is a procedure calling standard that can be used by most of the currently known programming languages. However, the generality of the standard means that, for some languages, a more efficient calling standard is possible; in this case the toolchain vendor has a choice between either using CPCS for maximum interoperability or implementing some other, more efficient, procedure calling standard at the expense of disallowing the toolchain from supporting some of the more elaborate programming languages. Of course, it is also possible to implement a toolchain that allows the user to choose whether CPCS or some other procedure calling standard is used.

The Common Procedure Calling Standard:

- Can be used for writing procedures with both constant and variable number of parameters.
- Allows parameters and return values of certain types to be passed in and out of the procedure in registers, thus frequently eliminating the need for using activation stack.
- Supports languages with static scoping rules (such as Pascal, Ada or SPL).
- Allows for nonlocal jumps and dynamic exception propagation (such as required by Pascal, Ada, SPL or C++).

## 3.1 Register usage conventions

As part of the CPCS, there are conventions about how registers are used across procedure calls. These conventions are outlined in the following sections.

### 3.1.1 Register-passable types

An important concept involved in the CPCS register usage conventions is that of a register-passable type. On an intuitive level, a register-passable type is a type whose values can be passed from caller to callee and/or back in a single Cereon register.

Only following types (and types derived from these types) are register-passable:

- `integer*1`, `integer*2`, `integer*4` and `integer*8`. Values of types `integer*1`, `integer*2`, `integer*4` are sign-extended to 64 bits when passed to/from a procedure in a register.
- `cardinal*1`, `cardinal*2`, `cardinal*4` and `cardinal*8`. Values of types `cardinal*1`, `cardinal*2`, `cardinal*4` are zero-extended to 64 bits when passed to/from a procedure in a register.
- `character*1`, `character*2` and `character*4`. Values of these types are zero-extended to 64 bits when passed to/from a procedure in a register.
- `real*4` and `real*8`. Values of the `real*4` type are converted to `real*8` when passed to/from a procedure in a register.
- `pointer` and `^T`, where `T` can be any type.
- `boolean`. Values of this type are zero-extended to 64 bits when passed to/from a procedure in a register.

- All enumerated types. Values of these types are zero-extended to 64 bits when passed to/from a procedure in a register.

### 3.1.2 Caller-saved registers

The following registers are not expected to keep their values across procedure calls:

- `$rv/$frv`.
- `$a0/$fa0 .. $a3/$fa3`.
- `$t0/$ft0 .. $t7/$ft7`.

These registers are typically used as temporary registers within local code sequences. If the caller needs any of these registers to keep their value after a procedure call, it must save them to memory before the call and reload after the callee has returned.

### 3.1.3 Callee-saved registers

The following registers are expected to keep their values across procedure calls:

- `$s0/$sf0 .. $s12/$fs12`.

These registers are typically used to store long-lived values, such as local variables. If the callee needs to use any of these registers, it must save them to memory before use and reload just before return.

### 3.1.4 Special registers

Several registers have special meaning under CPCS. These registers are:

- `$ra` – upon procedure entry this register contains an address of the instruction to which a jump must be made in order to perform procedure return. Typically a procedure will immediately save this register into its activation frame; however, in a leaf procedure (i.e. procedure that does not call any other procedures) this is optional. Note that the caller does not expect the `$ra` register to retain its value across procedure call.
- `$sp` – at any moment during program execution this register contains the address of the lowest memory byte occupied by the activation stack. Under CPCS stack starts at higher addresses and grows down. Both stack bottom and stack top addresses are always multiples of 8. The caller expects `$sp` to retain its value across calls of procedures with fixed number of parameters but not variadic procedures (see more in "Parameter passing" section below).
- `$fp` – the frame pointer register contains a pointer to a fixed location within the activation stack frame of the currently executing procedure. When a procedure is called, it saves the caller's `$fp` into its own activation frame before establishing an activation frame of its own. The caller expects the `$fp` register to retain its value across procedure calls.
- `$dp` – the display pointer register contains the frame pointer of the current procedure's immediately lexically enclosing parent. If the language in question does not permit nested procedures (which is the case in C or C++), the `$dp` register is always zero. If the language permits nested procedures

(such as SPL or Pascal), the `$dp` register will be zero whenever an outer-level procedure is being executed.

- `$gp` – this register contains an address of an unwind handler of the currently executing procedure. An unwind handler is a fragment of code that must be called if the activation frame of the current procedure is forcibly removed from an activation stack (this happens during C++ exception propagation or SPL nonlocal jumps). If the current procedure does not have an unwind handler, this register will be 0.

## 3.2 Parameter passing

When preparing to call a procedure, parameters are passed using argument registers `$a0/$fa0` .. `$a3/$fa3` and activation stack. The following rules are followed:

- The leftmost 4 parameters of register-passable types are passed in registers `$a0/$fa0` .. `$a3/$fa3` in that order (i.e. the 1st parameter of a register-passable type goes to `$a0/$fa0`, etc.) Note that these parameters are not necessarily consecutive; for example when calling a procedure that has the signature `(* : integer, * : label, * : real)` the 1st parameter will be passed in `$a0` and the 3rd in `$a1`, whereas the 2nd parameter will be passed on the stack. Note that `out` and `in out` parameters are technically pointers, so they are always register-passable.
- All parameters that are not passed in registers are passed on the stack. Parameters are pushed on the stack in right-to-left order, with each parameter being aligned at an 8-byte boundary.
- When a procedure with a fixed number of parameters is called, this procedure is expected to remove parameters from the stack.
- When a procedure with a variable number of parameters is called, this procedure is not expected to remove parameters from the stack, as it does not have sufficient information to do so. In this case, the `$sp` after the call is expected to be the same as `$sp` before the call, the caller will then need to adjust `$sp` before continuing.

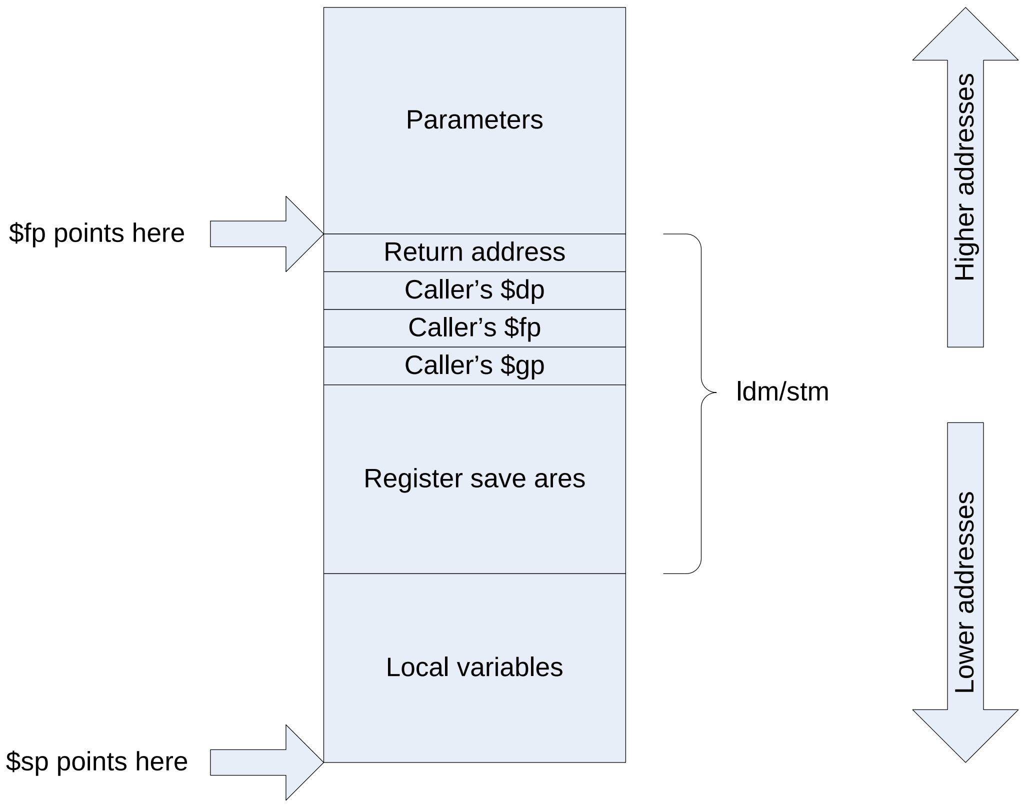
## 3.3 Returning values

If a procedure is a function (i.e. it returns an explicit value), the following rules are used:

- If a procedure returns value of a register-passable type, this value is returned in `$rv/$frv`.
- If a procedure returns value of any other type `T`, an implicit 0th parameter of type `^T` is assumed. Since pointers are register-passable, this implicit parameter will always be passed to the procedure in `$a0` (with the 1st "real" register-passable parameter shifted to `$a1/$fa1`, and so on). This parameter is a pointer to a value of type `T` allocated by the caller, where the return value must be stored.

## 3.4 Activation frame layout

The following diagram illustrates the layout of a single procedure activation frame:




Individual areas within the activation frame have the following functions:

- Parameters – this is the area used by those parameters which did not make it into registers. If all parameters are passed in registers, this area will be empty/
- Return address – the $ra register is saved here upon procedure entry. It will be restored just before returning to the caller in order to determine where to return.
- Caller's `$dp`, `$fp` and `$gp` – these 3 registers are saved here upon entry to the procedure. They will be restored just before returning to the caller, as the latter expects all three registers to retain their values across procedure calls.
- Register save area – if the procedure wishes to use any of the callee-saved registers for its own needs, these registers must be saved upon procedure entry and restored before returning to the caller. This is the area where these registers are saved.
- Local variables – if a procedure needs some in-memory local variables for its own needs, they are allocated here.

Note that the entire memory area marked as "`ldm`/`stm`" can be saved (upon procedure entry) or reloaded (upon procedure exit) by a single `ldm`/`stm` instruction. This ensures that procedure entry/exit code is but a few instructions long.

## 3.5 Accessing nonlocal variables

While in-memory local variables allow for straightforward access (via `$fp`-based relative addresses), accessing nonlocal variables from enclosing procedures is only slightly more complicated.

The central point of nonlocal variable accesses is in that it is always known at compile-time which level of nesting each procedure is declared at. Accessing a variable from an immediately enclosing procedure requires a `$dp`-based relative address instead of an `$fp`-based one. If the difference in the level of nesting is greater than 1, `$dp` points to an activation frame of an immediately enclosing procedure, and the static chain of long words at offset -8 within each activation frame gives an address of the next lexically enclosing activation frame.

## 3.6 Forced stack unwinding

In a language that permits exception propagation (such as C++) or nonlocal jumps (such as SPL) a situation may arise when a number of activation frames must be forcefully removed.

To unwind a single activation frame, the following steps must be performed:

1. An `ldm` from a register save area makes sure callee-saved registers have the same values they did before the call.
2. The unwind handler of the current procedure must be called, provided one exists. The address of an unwind handler of the procedure associated with the topmost activation frame is always in `$gp`; an absence of an unwind handler is represented by `$gp` = 0.
3. `$sp` is adjusted to `$fp` (for variadic procedures) or `$fp` + <parameter area size> (for procedures with fixed number of parameters).

After the steps above have been completed, the program state is restored to exactly what it was just before the call. Any number of stack frames can be unwound by repeating the same sequence of actions once per activation stack frame.

# 4 Nested procedure calling standard

The Nested Procedure Calling Standard (abbreviated henceforth as NPCS) is a procedure calling standard that can be used by programming languages that allow nonlocal nonstatic scopes and nested procedures but *not* the forced stack unwinding (such as Modula-2). It will typically be used by Modula-2 – only toolchains that do not care much about interoperability with other languages.

The Nested Procedure Calling Standard:

- Can be used for writing procedures with both constant and variable number of parameters.
- Allows parameters and return values of certain types to be passed in and out of the procedure in registers, thus frequently eliminating the need for using activation stack.
- Supports languages with static scoping rules (such as Pascal, Ada or SPL).
- Does not allow for nonlocal jumps and dynamic exception propagation.

## 4.1 Register usage conventions

As part of the NPCS, there are conventions about how registers are used across procedure calls. These conventions are outlined in the following sections.

### 4.1.1 Register-passable types

An important concept involved in the NPCS register usage conventions is that of a register-passable type. On an intuitive level, a register-passable type is a type whose values can be passed from caller to callee and/or back in a single Cereon register.

Only following types (and types derived from these types) are register-passable:

- `integer*1`, `integer*2`, `integer*4` and `integer*8`. Values of types `integer*1`, `integer*2`, `integer*4` are sign-extended to 64 bits when passed to/from a procedure in a register.
- `cardinal*1`, `cardinal*2`, `cardinal*4` and `cardinal*8`. Values of types `cardinal*1`, `cardinal*2`, `cardinal*4` are zero-extended to 64 bits when passed to/from a procedure in a register.
- `character*1`, `character*2` and `character*4`. Values of these types are zero-extended to 64 bits when passed to/from a procedure in a register.
- `real*4` and `real*8`. Values of the `real*4` type are converted to `real*8` when passed to/from a procedure in a register.
- `pointer` and `^T`, where `T` can be any type.
- `boolean`. Values of this type are zero-extended to 64 bits when passed to/from a procedure in a register.
- All enumerated types. Values of these types are zero-extended to 64 bits when passed to/from a procedure in a register.

### 4.1.2 Caller-saved registers

The following registers are not expected to keep their values across procedure calls:

- `$rv/$frv`.
- `$a0/$fa0 .. $a3/$fa3`.
- `$t0/$ft0 .. $t7/$ft7`.

These registers are typically used as temporary registers within local code sequences. If the caller needs any of these registers to keep their value after a procedure call, it must save them to memory before the call and reload after the callee has returned.

### 4.1.3 Callee-saved registers

The following registers are expected to keep their values across procedure calls:

- `$s0/$sf0 .. $s12/$fs12`.

These registers are typically used to store long-lived values, such as local variables. If the callee needs to use any of these registers, it must save them to memory before use and reload just before return.

### 4.1.4 Special registers

Several registers have special meaning under NPCS. These registers are:

- `$ra` – upon procedure entry this register contains an address of the instruction to which a jump must be made in order to perform procedure return. Typically a procedure will immediately save this register into its activation frame; however, in a leaf procedure (i.e. procedure that does not call any other procedures) this is optional. Note that the caller does not expect the `$ra` register to retain its value across procedure call.
- `$sp` – at any moment during program execution this register contains the address of the lowest memory byte occupied by the activation stack. Under NPCS stack starts at higher addresses and grows down. Both stack bottom and stack top addresses are always multiples of 8. The caller expects `$sp` to retain its value across calls of procedures with fixed number of parameters but not variadic procedures (see more in "Parameter passing" section below).
- `$fp` – the frame pointer register contains a pointer to a fixed location within the activation stack frame of the currently executing procedure. When a procedure is called, it saves the caller's `$fp` into its own activation frame before establishing an activation frame of its own. The caller expects the `$fp` register to retain its value across procedure calls.
- `$dp` – the display pointer register contains the frame pointer of the current procedure's immediately lexically enclosing parent. If the language in question does not permit nested procedures (which is the case in C or C++), the `$dp` register is always zero. If the language permits nested procedures (such as SPL or Pascal), the `$dp` register will be zero whenever an outer-level procedure is being executed.

## 4.2 Parameter passing

When preparing to call a procedure, parameters are passed using argument registers `$a0/$fa0` .. `$a3/$fa3` and activation stack. The following rules are followed:

- The leftmost 4 parameters of register-passable types are passed in registers `$a0/$fa0` .. `$a3/$fa3` in that order (i.e. the 1st parameter of a register-passable type goes to `$a0/$fa0`, etc.) Note that these parameters are not necessarily consecutive; for example when calling a procedure that has the signature `(* : integer, * : label, * : real)` the 1st parameter will be passed in `$a0` and the 3rd in `$a1`, whereas the 2nd parameter will be passed on the stack. Note that `out` and `in out` parameters are technically pointers, so they are always register-passable.
- All parameters that are not passed in registers are passed on the stack. Parameters are pushed on the stack in right-to-left order, with each parameter being aligned at an 8-byte boundary.
- When a procedure with a fixed number of parameters is called, this procedure is expected to remove parameters from the stack.
- When a procedure with a variable number of parameters is called, this procedure is not expected to remove parameters from the stack, as it does not have sufficient information to do so. In this case, the `$sp` after the call is expected to be the same as `$sp` before the call, the caller will then need to adjust `$sp` before continuing.

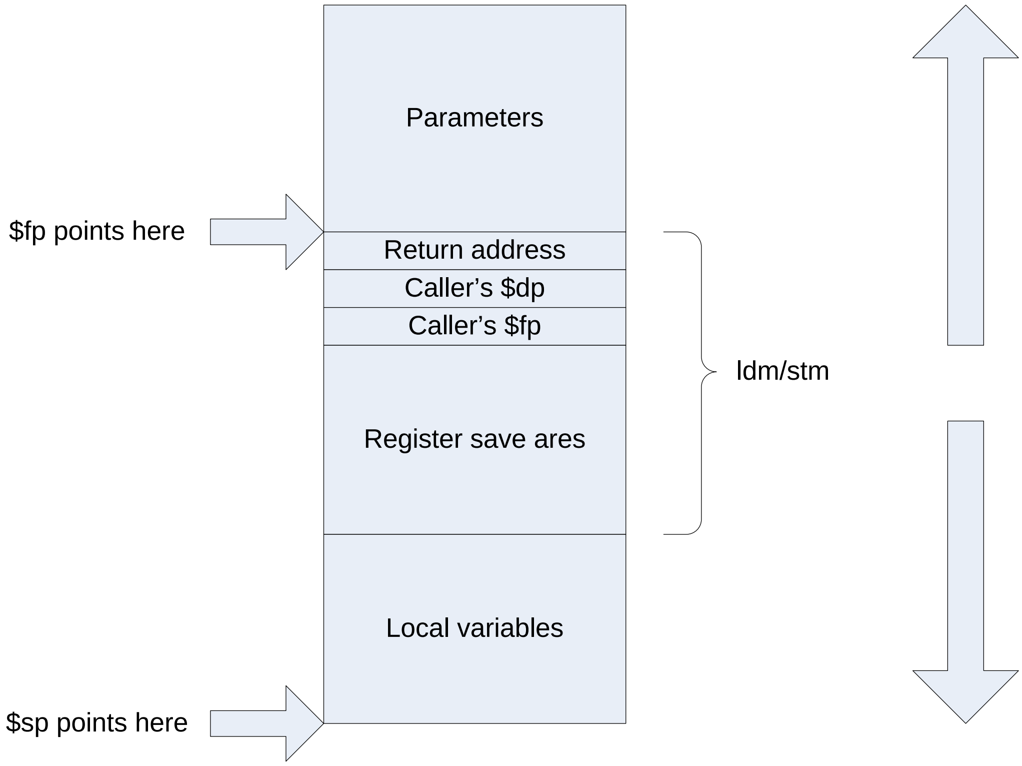
## 4.3 Returning values

If a procedure is a function (i.e. it returns an explicit value), the following rules are used:

- If a procedure returns value of a register-passable type, this value is returned in `$rv/$frv`.
- If a procedure returns value of any other type `T`, an implicit 0th parameter of type `^T` is assumed. Since pointers are register-passable, this implicit parameter will always be passed to the procedure in `$a0` (with the 1st "real" register-passable parameter shifted to `$a1/$fa1`, and so on). This parameter is a pointer to a value of type `T` allocated by the caller, where the return value must be stored.

## 4.4 Activation frame layout

The following diagram illustrates the layout of a single procedure activation frame:




Individual areas within the activation frame have the following functions:

- Parameters – this is the area used by those parameters which did not make it into registers. If all parameters are passed in registers, this area will be empty/
- Return address – the $ra register is saved here upon procedure entry. It will be restored just before returning to the caller in order to determine where to return.
- Caller’s `$dp` and `$fp` – these 2 registers are saved here upon entry to the procedure. They will be restored just before returning to the caller, as the latter expects both registers to retain their values across procedure calls.
- Register save area – if the procedure wishes to use any of the callee-saved registers for its own needs, these registers must be saved upon procedure entry and restored before returning to the caller. This is the area where these registers are saved.
- Local variables – if a procedure needs some in-memory local variables for its own needs, they are allocated here.

Note that the entire memory area marked as “`ldm`/`stm`” can be saved (upon procedure entry) or reloaded (upon procedure exit) by a single `ldm`/`stm` instruction. This ensures that procedure entry/exit code is but a few instructions long.

## 4.5 Accessing nonlocal variables

While in-memory local variables allow for straightforward access (via `$fp`-based relative addresses), accessing nonlocal variables from enclosing procedures is only slightly more complicated.

The central point of nonlocal variable accesses is in that it is always known at compile-time which level of nesting each procedure is declared at. Accessing a variable from an immediately enclosing procedure requires a `$dp`-based relative address instead of an `$fp`-based one. If the difference in the level of nesting is greater than 1, `$dp` points to an activation frame of an immediately enclosing procedure, and the static chain of long words at offset -8 within each activation frame gives an address of the next lexically enclosing activation frame.

# 5 Throwing procedure calling standard

The Throwing  Procedure Calling Standard (abbreviated henceforth as TPCS) is a procedure calling standard that can be used by programming languages that allow forced stack unwinding and exception propagation but do not allow nested procedures (such as C and C++). It will typically be used by C/C++ – only toolchains that do not care much about interoperability with other languages.

The Throwing Procedure Calling Standard:

- Can be used for writing procedures with both constant and variable number of parameters.
- Allows parameters and return values of certain types to be passed in and out of the procedure in registers, thus frequently eliminating the need for using activation stack.
- Does not supports languages with nonlocal static scoping rules.
- Allows for nonlocal jumps and dynamic exception propagation (such as required by C or C++).

## 5.1 Register usage conventions

As part of the TPCS, there are conventions about how registers are used across procedure calls. These conventions are outlined in the following sections.

### 5.1.1 Register-passable types

An important concept involved in the TPCS register usage conventions is that of a register-passable type. On an intuitive level, a register-passable type is a type whose values can be passed from caller to callee and/or back in a single Cereon register.

Only following types (and types derived from these types) are register-passable:

- `integer*1`, `integer*2`, `integer*4` and `integer*8`. Values of types `integer*1`, `integer*2`, `integer*4` are sign-extended to 64 bits when passed to/from a procedure in a register.
- `cardinal*1`, `cardinal*2`, `cardinal*4` and `cardinal*8`. Values of types `cardinal*1`, `cardinal*2`, `cardinal*4` are zero-extended to 64 bits when passed to/from a procedure in a register.
- `character*1`, `character*2` and `character*4`. Values of these types are zero-extended to 64 bits when passed to/from a procedure in a register.
- `real*4` and `real*8`. Values of the `real*4` type are converted to `real*8` when passed to/from a procedure in a register.
- `pointer` and `^T`, where `T` can be any type.
- `boolean`. Values of this type are zero-extended to 64 bits when passed to/from a procedure in a register.
- All enumerated types. Values of these types are zero-extended to 64 bits when passed to/from a procedure in a register.

### 5.1.2 Caller-saved registers

The following registers are not expected to keep their values across procedure calls:

- `$rv/$frv`.
- `$a0/$fa0 .. $a3/$fa3`.
- `$t0/$ft0 .. $t7/$ft7`.

These registers are typically used as temporary registers within local code sequences. If the caller needs any of these registers to keep their value after a procedure call, it must save them to memory before the call and reload after the callee has returned.

### 5.1.3 Callee-saved registers

The following registers are expected to keep their values across procedure calls:

- `$s0/$sf0 .. $s12/$fs12`.

These registers are typically used to store long-lived values, such as local variables. If the callee needs to use any of these registers, it must save them to memory before use and reload just before return.

### 5.1.4 Special registers

Several registers have special meaning under TPCS. These registers are:

- `$ra` – upon procedure entry this register contains an address of the instruction to which a jump must be made in order to perform procedure return. Typically a procedure will immediately save this register into its activation frame; however, in a leaf procedure (i.e. procedure that does not call any other procedures) this is optional. Note that the caller does not expect the `$ra` register to retain its value across procedure call.
- `$sp` – at any moment during program execution this register contains the address of the lowest memory byte occupied by the activation stack. Under TPCS stack starts at higher addresses and grows down. Both stack bottom and stack top addresses are always multiples of 8. The caller expects `$sp` to retain its value across calls of procedures with fixed number of parameters but not variadic procedures (see more in “Parameter passing” section below).
- `$fp` – the frame pointer register contains a pointer to a fixed location within the activation stack frame of the currently executing procedure. When a procedure is called, it saves the caller’s `$fp` into its own activation frame before establishing an activation frame of its own. The caller expects the `$fp` register to retain its value across procedure calls.
- `$gp` – this register contains an address of an unwind handler of the currently executing procedure. An unwind handler is a fragment of code that must be called if the activation frame of the current procedure is forcibly removed from an activation stack (this happens during C++ exception propagation or SPL nonlocal jumps). If the current procedure does not have an unwind handler, this register will be 0.

## 5.2 Parameter passing

When preparing to call a procedure, parameters are passed using argument registers `$a0/$fa0` .. `$a3/$fa3` and activation stack. The following rules are followed:

- The leftmost 4 parameters of register-passable types are passed in registers `$a0/$fa0` .. `$a3/$fa3` in that order (i.e. the 1st parameter of a register-passable type goes to `$a0/$fa0`, etc.) Note that these parameters are not necessarily consecutive; for example when calling a procedure that has the signature `(* : integer, * : label, * : real)` the 1st parameter will be passed in `$a0` and the 3rd in `$a1`, whereas the 2nd parameter will be passed on the stack. Note that `out` and `in out` parameters are technically pointers, so they are always register-passable.
- All parameters that are not passed in registers are passed on the stack. Parameters are pushed on the stack in right-to-left order, with each parameter being aligned at an 8-byte boundary.
- When a procedure with a fixed number of parameters is called, this procedure is expected to remove parameters from the stack.
- When a procedure with a variable number of parameters is called, this procedure is not expected to remove parameters from the stack, as it does not have sufficient information to do so. In this case, the `$sp` after the call is expected to be the same as `$sp` before the call, the caller will then need to adjust `$sp` before continuing.

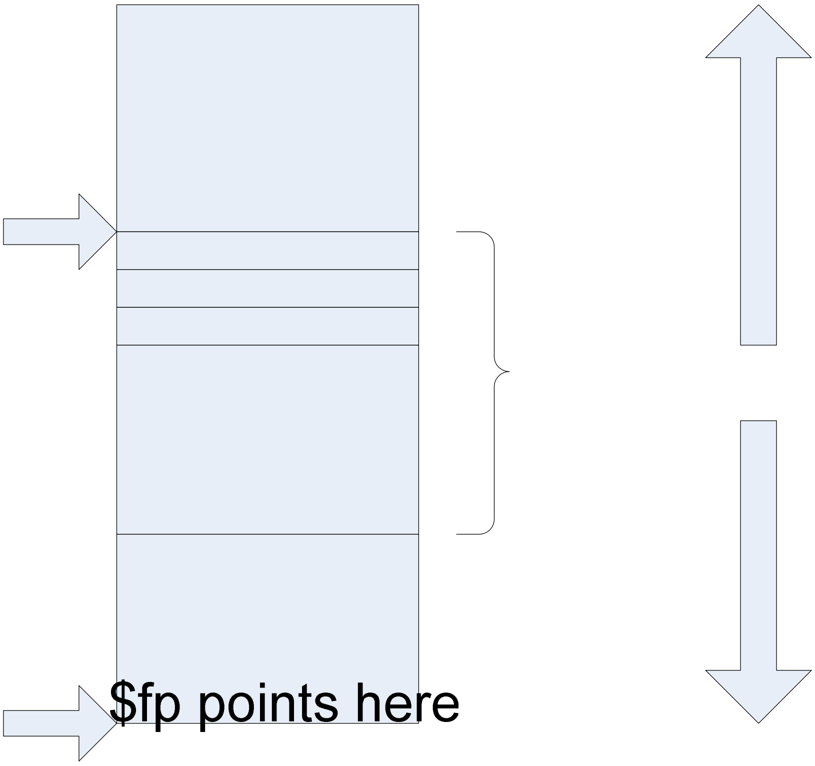
## 5.3 Returning values

If a procedure is a function (i.e. it returns an explicit value), the following rules are used:

- If a procedure returns value of a register-passable type, this value is returned in `$rv/$frv`.
- If a procedure returns value of any other type `T`, an implicit 0th parameter of type `^T` is assumed. Since pointers are register-passable, this implicit parameter will always be passed to the procedure in `$a0` (with the 1st "real" register-passable parameter shifted to `$a1/$fa1`, and so on). This parameter is a pointer to a value of type `T` allocated by the caller, where the return value must be stored.

## 5.4 Activation frame layout

The following diagram illustrates the layout of a single procedure activation frame:




Individual areas within the activation frame have the following functions:

- Parameters – this is the area used by those parameters which did not make it into registers. If all parameters are passed in registers, this area will be empty/
- Return address – the $ra register is saved here upon procedure entry. It will be restored just before returning to the caller in order to determine where to return.
- Caller's `$dp`, `$fp` and `$gp` – these 3 registers are saved here upon entry to the procedure. They will be restored just before returning to the caller, as the latter expects all three registers to retain their values across procedure calls.
- Register save area – if the procedure wishes to use any of the callee-saved registers for its own needs, these registers must be saved upon procedure entry and restored before returning to the caller. This is the area where these registers are saved.
- Local variables – if a procedure needs some in-memory local variables for its own needs, they are allocated here.

Note that the entire memory area marked as "`ldm`/`stm`" can be saved (upon procedure entry) or reloaded (upon procedure exit) by a single `ldm`/`stm` instruction. This ensures that procedure entry/exit code is but a few instructions long.

## 5.5 Forced stack unwinding

In a language that permits exception propagation (such as C++) or nonlocal jumps (such as SPL) a situation may arise when a number of activation frames must be forcefully removed.

To unwind a single activation frame, the following steps must be performed:

4. An `ldm` from a register save area makes sure callee-saved registers have the same values they did before the call.
5. The unwind handler of the current procedure must be called, provided one exists. The address of an unwind handler of the procedure associated with the topmost activation frame is always in `$gp`; an absence of an unwind handler is represented by `$gp` = 0.
6. `$sp` is adjusted to `$fp` (for variadic procedures) or `$fp` + <parameter area size> (for procedures with fixed number of parameters).

After the steps above have been completed, the program state is restored to exactly what it was just before the call. Any number of stack frames can be unwound by repeating the same sequence of actions once per activation stack frame.

# 6 Basic procedure calling standard

The Basic Procedure Calling Standard (abbreviated henceforth as BPCS) is a procedure calling standard that can be used by programming languages that do not allow nested procedures or forced stack unwinding (such as COBOL or XPL) and is also useful in Assembler-only programs. Although the least general standard, the BPCS allows the most efficient procedure call/return sequences to be implemented. It will typically be used by low-level toolchains that do not care much about interoperability with other languages, preferring to address the maximum operation efficiency instead.

The Basic Procedure Calling Standard:

- Can be used for writing procedures with both constant and variable number of parameters.
- Allows parameters and return values of certain types to be passed in and out of the procedure in registers, thus frequently eliminating the need for using activation stack.
- Does not support languages with nonlocal static scoping rules.
- Does not allow for nonlocal jumps and dynamic exception propagation.

## 6.1 Register usage conventions

As part of the BPCS, there are conventions about how registers are used across procedure calls. These conventions are outlined in the following sections.

### 6.1.1 Register-passable types

An important concept involved in the BPCS register usage conventions is that of a register-passable type. On an intuitive level, a register-passable type is a type whose values can be passed from caller to callee and/or back in a single Cereon register.

Only following types (and types derived from these types) are register-passable:

- `integer*1`, `integer*2`, `integer*4` and `integer*8`. Values of types `integer*1`, `integer*2`, `integer*4` are sign-extended to 64 bits when passed to/from a procedure in a register.
- `cardinal*1`, `cardinal*2`, `cardinal*4` and `cardinal*8`. Values of types `cardinal*1`, `cardinal*2`, `cardinal*4` are zero-extended to 64 bits when passed to/from a procedure in a register.
- `character*1`, `character*2` and `character*4`. Values of these types are zero-extended to 64 bits when passed to/from a procedure in a register.
- `real*4` and `real*8`. Values of the `real*4` type are converted to `real*8` when passed to/from a procedure in a register.
- `pointer` and `^T`, where `T` can be any type.
- `boolean`. Values of this type are zero-extended to 64 bits when passed to/from a procedure in a register.
- All enumerated types. Values of these types are zero-extended to 64 bits when passed to/from a procedure in a register.

### 6.1.2 Caller-saved registers

The following registers are not expected to keep their values across procedure calls:

- `$rv/$frv`.
- `$a0/$fa0 .. $a3/$fa3`.
- `$t0/$ft0 .. $t7/$ft7`.

These registers are typically used as temporary registers within local code sequences. If the caller needs any of these registers to keep their value after a procedure call, it must save them to memory before the call and reload after the callee has returned.

### 6.1.3 Callee-saved registers

The following registers are expected to keep their values across procedure calls:

- `$s0/$sf0 .. $s12/$fs12`.

These registers are typically used to store long-lived values, such as local variables. If the callee needs to use any of these registers, it must save them to memory before use and reload just before return.

### 6.1.4 Special registers

Several registers have special meaning under BPCS. These registers are:

- `$ra` – upon procedure entry this register contains an address of the instruction to which a jump must be made in order to perform procedure return. Typically a procedure will immediately save this register into its activation frame; however, in a leaf procedure (i.e. procedure that does not call any other procedures) this is optional. Note that the caller does not expect the `$ra` register to retain its value across procedure call.
- `$sp` – at any moment during program execution this register contains the address of the lowest memory byte occupied by the activation stack. Under BPCS stack starts at higher addresses and grows down. Both stack bottom and stack top addresses are always multiples of 8. The caller expects `$sp` to retain its value across calls of procedures with fixed number of parameters but not variadic procedures (see more in "Parameter passing" section below).
- `$fp` – the frame pointer register contains a pointer to a fixed location within the activation stack frame of the currently executing procedure. When a procedure is called, it saves the caller's `$fp` into its own activation frame before establishing an activation frame of its own. The caller expects the `$fp` register to retain its value across procedure calls.

## 6.2 Parameter passing

When preparing to call a procedure, parameters are passed using argument registers `$a0/$fa0` .. `$a3/$fa3` and activation stack. The following rules are followed:

- The leftmost 4 parameters of register-passable types are passed in registers `$a0/$fa0` .. `$a3/$fa3` in that order (i.e. the 1st parameter of a register-passable type goes to `$a0/$fa0`, etc.) Note that these parameters are not necessarily consecutive; for example when calling a procedure that has the signature `(* : integer, * : label, * : real)` the 1st parameter will be passed in `$a0` and the 3rd in `$a1`, whereas the 2nd parameter will be passed on the stack. Note that `out` and `in out` parameters are technically pointers, so they are always register-passable.

- All parameters that are not passed in registers are passed on the stack. Parameters are pushed on the stack in right-to-left order, with each parameter being aligned at an 8-byte boundary.
- When a procedure with a fixed number of parameters is called, this procedure is expected to remove parameters from the stack.
- When a procedure with a variable number of parameters is called, this procedure is not expected to remove parameters from the stack, as it does not have sufficient information to do so. In this case, the `$sp` after the call is expected to be the same as `$sp` before the call, the caller will then need to adjust `$sp` before continuing.

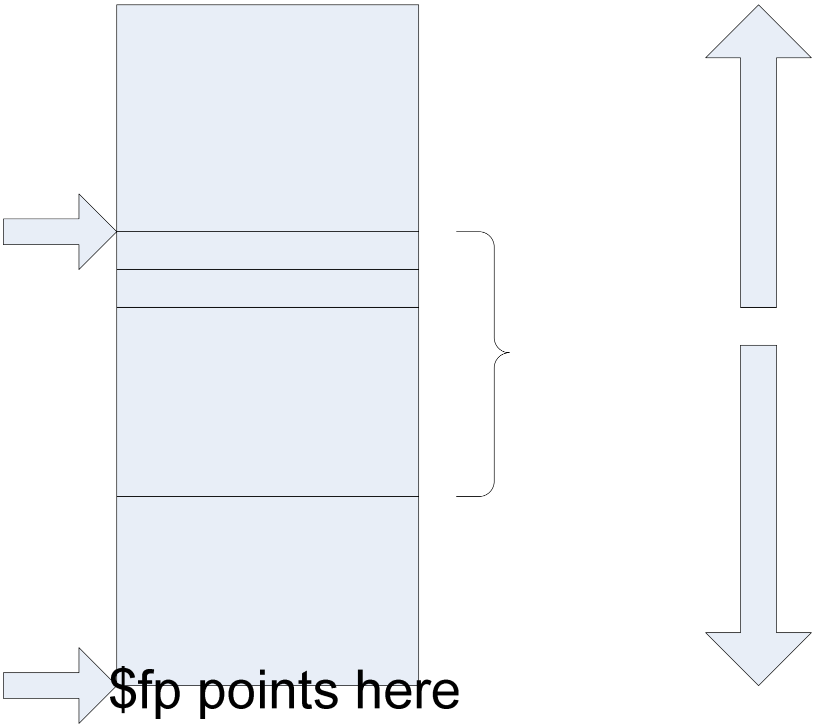
## 6.3 Returning values

If a procedure is a function (i.e. it returns an explicit value), the following rules are used:

- If a procedure returns value of a register-passable type, this value is returned in `$rv/$frv`.
- If a procedure returns value of any other type `T`, an implicit 0th parameter of type `^T` is assumed. Since pointers are register-passable, this implicit parameter will always be passed to the procedure in `$a0` (with the 1st "real" register-passable parameter shifted to `$a1/$fa1`, and so on). This parameter is a pointer to a value of type `T` allocated by the caller, where the return value must be stored.

## 6.4 Activation frame layout

The following diagram illustrates the layout of a single procedure activation frame:




Individual areas within the activation frame have the following functions:

- Parameters – this is the area used by those parameters which did not make it into registers. If all parameters are passed in registers, this area will be empty/
- Return address – the $ra register is saved here upon procedure entry. It will be restored just before returning to the caller in order to determine where to return.
- Caller's `$fp` – this register is saved here upon entry to the procedure. It will be restored just before returning to the caller, as the latter expects the register to retain its values across procedure calls.
- Register save area – if the procedure wishes to use any of the callee-saved registers for its own needs, these registers must be saved upon procedure entry and restored before returning to the caller. This is the area where these registers are saved.
- Local variables – if a procedure needs some in-memory local variables for its own needs, they are allocated here.

Note that the entire memory area marked as "`ldm/stm`" can be saved (upon procedure entry) or reloaded (upon procedure exit) by a single `ldm/stm` instruction. This ensures that procedure entry/exit code is but a few instructions long.

# 7 Appendix A: GNU Free Documentation License

Version 1.2, November 2002



**0. PREAMBLE**

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

**1. APPLICABILITY AND DEFINITIONS**

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

**2. VERBATIM COPYING**

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

**3. COPYING IN QUANTITY**

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

**4. MODIFICATIONS**

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.
- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

**5. COMBINING DOCUMENTS**

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

**6. COLLECTIONS OF DOCUMENTS**

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

**7. AGGREGATION WITH INDEPENDENT WORKS**

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

**8. TRANSLATION**

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.