Table of Contents

Notices....2
Introduction....2
How to Use This Document....3
ACALL....4
AFUN....5
CALL....6
CALLR....8
FRAME....10
FUNCTION....13
ICALL....15
IFUN....17
LOCAL....19
RETURN....20
SCALL....22
SFUN....24
STACK....25
STKINIT....26
HOWTO use Functions....28
Why Functions?....28
How the Stack Works....30
Calling a Function....31
Register Usage....31
Function Prolog....33
Function Body....33
Function Epilog....34
The Stack Frame....34
FRAME....35
General Registers....36
Floating Point Registers....36
Back Pointer and Compiler Reserved Field....37
LOCAL....37
Initializing the Stack....38
STACK....38
STKINIT....39
Function Definition....39
FUNCTION....39
RETURN....39
Using Functions....40
CALL....40
CALLR....40
Structuring the Assembly....42
Appendix A – SATK Functions vs. ELF ABI....43

Register Usage....44
R2-R5....45
R12....45
R13....46
R14....46
R15....46
A0, A1....46
Appendix B – SATK Functions vs. Legacy Subroutines....47
Register Usage....47
Parameter Passing / Returning Results....47
Register Save Areas vs. Stack Frame....48

Notices

IBM is a registered trademark of International Business Machines Corporation.

Introduction

This manual describes a set of macros and their usage in implementing a standard routine calling conventions.

The following macros found in the maclib macro library directory belong to the func category.

Macro	Description
ACALL	Call an architecture specific function by its architecture independent name
AFUN	Initiate an architecture specific function definition with its architecture independent name.
CALL	Call a function by its defined name
CALLR	Call a function by its register resident location.
FRAME	Define the SATK function usage of the stack frame for the assembly
FUNCTION	Initiate a function's definition.
ICALL	Call an input/output architecture specific function by its architecture independent name
IFUN	Initiate an input/output architecture specific function definition with its architecture independent name.

Macro	Description
LOCAL	Define local usage of the stack frame by the next defined function
RETURN	Leave a function, returning control to the function's caller
STACK	Identify the “bottom” of the function stack
STKINIT	Initialize the stack for its first stack frame.

These dummy sections may be created by the FRAME macro depending upon the register environment.

DSECT	Description
STKF	Define function usage of a stack frame in a 32-bit register environment
STKG	Define function usage of a stack frame in a 64-bit register environment

How to Use This Document

If this is the first time you are reading this document, it is recommended that you read the “HOWTO...” section first. This gives the background helpful in understanding where each macro fits into “functions” as implemented by the Stand Alone Tool Kit (SATK).

For information related to architecture levels, refer to either SATK.odt or SATK.pdf. Architecture levels apply to nearly all macros.

ACALL

Source file: maclib/ACALL.mac

Macro Format:

```
[label]   ACALL func
                [,INLINE=[A|J]]
```

The ACALL macro transfers program control to the a function performing architecture specific function.

Assembly Considerations:

- The same as for the CALL macro.

Execution Considerations:

- The same as for the CALL macro.

Label Field Usage:

- The same as for the CALL macro.

Positional Parameters:

1. The func positional parameter identifies the architecture specific function being called. The parameter must identify a label that was established by a AFUN macro within the assembly.

Keyword Parameters:

- The same as for the CALL macro.

Programming Note:

An architecture specific suffix is appended to the label provided by the func positional parameter. Architecture level 1 uses the single character '1' appended to the function being called. Architecture level 2 uses the single character '2', etc.

For example, if the current assembly architecture level is 4 and the label field contains TIMER, the architecture specific function name will actually be TIMER4.

```
ACALL TIMER is equivalent to
CALL  TIMER4
```

AFUN

Source file: maclib/AFUN.mac

Macro Format:

```
label    AFUN  [FP=[YES|NO]
               [,AFP=[YES|NO]]
```

AFUN macro initiates a subroutine using the SATK standardized calling conventions performing architecture specific functions by the current assembly architecture level.

Assembly Considerations:

- The same as for the FUNCTION macro.

Execution Considerations:

- The same as for the FUNCTION macro.

Label Field Usage:

The label field is required. It establishes the entry point of the function and is used directly by the ACALL macro to call the function or indirectly by the CALLR macro by previously loading a register with the label, suffixed by its architecture specific suffix.

Positional Parameters: None

Keyword Parameters:

- The same as for the FUNCTION macro.

Programming Note:

An architecture specific suffix is appended to the label defining the function's start. Architecture level 1 uses the single character '1' appended to the function being called. Architecture level 2 uses the single character '2', etc.

For example, if the current assembly architecture level is 4 and the label field contains TIMER, the architecture specific function name will actually be TIMER4.

```
TIMER    AFUN  is equivalent to
TIMER4   FUNCTION
```

CALL

Source file: maclib/CALL.mac

Macro Format:

```
[label]    CALL  func
                 [,INLINE=[A|J]]
```

The CALL macro transfers program control to the specified function label. Depending upon the architecture the following registers may be used for passing arguments and can be ensured of being preserved upon returning from the called function. Function arguments must be placed in the argument registers before the CALL macro is executed.

Level	Type	Argument Registers	Return Registers	Preserved
1-9	GR	R2-R6	R2-R5	R6-R15
1-6	FP	F0, F2	F0, F2	F4, F6
7-9	FP	F0-F7	F0-F7	F8-F15

The same registers used for passing arguments to the function may be used by the function for returning results to the caller. Typically, though, only R2 or F0 are used for a single return value. Return registers not used by the called function have unpredictable content as do general registers 0 and 1 upon return to the caller.

Assembly Considerations:

- A preceding ARCHLVL or ARCHIND macro is required to establish the architectural environment in which the macro is expected to operate.
- An established base register is required by the macro preceding its use.

Execution Considerations:

- SATK function register conventions are in use.
- General register 15 must contain the address of the first stack frame used for the preservation of the program making the first level of function calls. The stack frame pointer, general register 15, must be established prior to executing the CALL macro by use of the STKINIT macro or some other method initializing the stack frame pointer, general register 15.

Label Field Usage:

The label field, if present, defines an assembly label associated with the first instruction generated by the macro.

Positional Parameters:

1. The func positional parameter identifies the function being called. The parameter must identify a label that was established by a FUNCTION macro within the assembly.

Keyword Parameters:

Keyword	Default	Description
INLINE	Note 1	Specifies how the function's address is determined. Values allowed: • A – an inline aligned address constant based upon the current architecture level is created, or • J – forces use of a positional relative long address.

Note 1. If omitted, the current architecture level will dictate the selection:

- Levels 0-7 default to A,
- Levels 8 or 9 default to J.

CALLR

Source file: maclib/CALLR.mac

Macro Format:

```
[label]   CALLR [([reg|1])]
```

The CALLR macro transfers program control to the function whose starting address has been placed in the specified general register. Depending upon the architecture the following registers may be used for passing arguments and can be ensured of being preserved upon returning from the called function. Function arguments and must be placed in the argument registers before the CALLR macro is executed.

Level	Type	Argument/Return Registers	Preserved
1-9	GR	R2-R4	R6-R15
1-6	FP	F0, F2	F4, F6
7-9	FP	F0-F7	F8-F15

The same registers used for passing arguments to the function may be used by the function for returning results to the caller. Typically, though, only R2 or F0 are used for a single return value. Return registers not used by the called function have unpredictable content as do general registers 0 and 1 upon return to the caller.

Assembly Considerations:

- A preceding ARCHLVL or ARCHIND macro is required to establish the operation synonyms used by the macro.

Execution Considerations:

- General register 15 must contain the address of the first stack frame used for the preservation of the program making the first level of function calls. The stack frame pointer, general register 15, must be established prior to executing the CALLR macro by use of the STKINIT macro or some other method initializing the stack frame pointer, general register 15.

Label Field Usage:

The label field, if present, defines an assembly label associated with the first instruction generated by the macro.

Positional Parameters:

1. The `reg` positional parameter identifies the register containing the address of the function being called. If omitted, general register 1 is assumed.

Keyword Parameters: None

FRAME

Source file: maclib/FRAME.mac

Macro Format:

```
FRAME [PACK=[YES|NO]]
      [,FP=[YES|NO]]
      [,AFP=[YES|NO]]
      [,BACKPTR=[YES|NO]]
      [,PRINT=[ON|OFF]]
```

The FRAME macro defines the SATK stack frame structure used by functions within the assembly. Either one of two dummy sections will be created depending upon the register size of the current architecture level:

- STKF for a 32-bit register CPU environment, or
- STKG for a 64-bit register CPU environment.

The various keyword parameters control the size of a function's stack frame used by this category of macros. The PACK keyword parameter allows restricting of available fields to only those registers requiring preservation by a called function. FP, AFP, and BACKPTR keyword parameters dictate whether additional fields are created. Which fields are created dictate which registers **can** be preserved by a function's prolog instruction sequence created by a function's FUNCTION macro. Which fields **are** actually preserved are influenced by the FUNCTION macro itself. The following table describes the number of registers that may be saved depending upon the keyword parameters and current architecture level.

Level	Type	Size	FP	AFP	BACKPTR	PACK=NO	PACK=YES
1-8	GR	4	--	--	--	14	10
1-8	Backptr/Compiler	4	--	--	NO	0	0
1-8	Backptr/Compiler	4	--	--	YES	2	2
1-9	FP	8	NO	NO	--	0	0
1-9	FP	8	YES	NO	--	4	2
7-9	FP	8	YES	YES	--	16	8
9	GR	8	--	--	--	14	10
9	Backptr/Compiler	8	--	--	NO	0	0
9	Backptr/Compiler	8	--	--	YES	2	2

See the corresponding table in the FUNCTION and RETURN macro descriptions for the details of which registers are saved and restored, respectively.

Assembly Considerations:

- A preceding ARCHLVL or ARCHIND macro is required to establish the expected architecture execution environment.
- Fields not provided by the definition may be replaced with local stack frame fields managed by the calling function that needs the register content preserved.
- The size of the initial stack frame, depending upon execution environment is given by subtracting the start of the stack frame dummy section, STKF or STKG, from the dummy section's STKFLCL or STKGLCL label, respectively

Execution Considerations:

- PACK=YES provides the smallest stack frame size, but, for debugging purposes, PACK=NO will provide the maximum amount of information.

Label Field Usage: Ignored

Positional Parameters: None

Keyword Parameters:

Keyword	Default	Description
AFP	NO	Specify: • YES to reserve stack frame space for the saving of the twelve additional floating point registers (requires FP=YES) or • NO to reserve no additional space in the minimum stack frame for any additional floating point registers.
BACKPTR	NO	Specify: • YES to reserve stack frame space for a back pointer and compiler field in the minimum stack frame or • NO to reserve no space for a back pointer or compiler field.
FP	NO	Specify: • YES to reserve space for the saving of FP0, FP2, FP4, and FP6 in the minimum stack frame; or • NO to reserve no space any floating point registers.
PACK	NO	Specify • YES – reserving stack frame fields for only preserved registers, or • NO – reserve stack frame space for most all general registers.
PRINT	none	Specify • ON to cause the current PRINT setting to be saved and force printing of the generated DSECT.

Keyword	Default	Description
		• OFF to cause the current PRINT setting to be save and inhibit printing of the generated DSECT. • Omit the keyword parameter to use the current PRINT setting. If either ON or OFF is specified, the current PRINT setting is restored after DSECT creation.

FUNCTION

Source file: maclib/FUNCTION.mac

Macro Format:

```
label     FUNCTION [FP=[YES|NO]
                   [,AFP=[YES|NO]]
```

The FUNCTION macro initiates a subroutine using the SATK standardized calling conventions. It performs the function prolog sequence. The function prolog:

- If preceded by a LOCAL macro, ends the definition the local stack frame fields, and positions the assembly at the point the LOCAL macro interrupted the current control section;
- Establishes the function's base register, always general register 13; and
- Preserves the calling function's state by saving it in the caller's stack frame.

The function, once initiated, continues until a RETURN macro is encountered. Within the function, other func category macros are restricted to CALL or CALLR macros. Other categories of macros or user defined macros may be used provided their respective usage requirements are satisfied and there register usage is consistent with that of a function.

Branching outside of the function is not recommended. Any explicit subroutine calls made from within a function must follow the register usage rules of a function, or preserve the function state. In general, it is recommended that only other functions be called from within a function.

The keyword parameters control whether the standard floating point or additional floating point registers are saved. As a prerequisite, the corresponding keyword parameters on the assembly's FRAME macro must have also been specified as YES.

If the FRAME macro had BACKPTR=YES specified, the function prolog will automatically save the back pointer. The compiler field is not used by function macros.

The following table identifies the registers saved by the function prolog depending upon architecture level, FUNCTION macro keywords and whether the FRAME macro had its keyword parameter PACK specified as YES or NO. The column “Available” identifies the registers by type that may be freely used by the function.

Level	Type	FP	AFP	Available	PACK=NO	PACK=YES
1-9	GR	--	--	R2-R12	R2-R15	R6-R15
1-9	FP	NO	NO	F0, F2, F4, F6	none	none
1-9	FP	YES	NO	F0, F2, F4, F6	F0, F2, F4, F6	F4, F6
7-9	FP	YES	YES	F0-F15	F0-F15	F8-F15

Assembly Considerations:

- A preceding ARCHLVL or ARCHIND macro is required to establish the expected architecture execution environment and operation synonyms.

Execution Considerations:

- Instruction sequences within the function must conform to function standard register usage.
- The CALL or CALLR macro must never be executed to call a function before a legitimate stack frame pointer has been established. See STACK and STKINIT macros.

Label Field Usage:

The label field is required. It establishes the entry point of the function and is used directly by the CALL macro to call the function or indirectly by the CALLR macro by previously loading a register with the label's address.

Positional Parameters: None

Keyword Parameters:

Keyword	Default	Description
AFP	NO	Specify: • YES to save the caller's additional floating point registers in the stack frame. Requires the assembly's FRAME macro to have specified FP=YES and AFP=YES. See the above table for the actual additional floating point registers that are saved. • NO inhibits the saving of the caller's additional floating point registers.
FP	NO	Specify: • YES to save the caller's standard of floating point registers in the stack frame. Requires the assembly's FRAME macro to have specified FP=YES. See the above table for the actual standard floating point registers that are saved. • NO inhibits the saving of the caller's standard floating point registers.

ICALL

Source file: maclib/ICALL.mac

Macro Format:

```
[label]   ICALL func
                [,INLINE=[A|J]]
```

The ICALL macro transfers program control to the a function performing I/O architecture specific functions.

Assembly Considerations:

- The same as for the CALL macro.

Execution Considerations:

- The same as for the CALL macro.

Label Field Usage:

- The same as for the CALL macro.

Positional Parameters:

1. The func positional parameter identifies the architecture specific function being called. The parameter must identify a label that was established by a IFUN macro within the assembly.

Keyword Parameters:

- The same as for the CALL macro.

Programming Note:

Input/output architecture function suffixes are assigned by architecture level:

- 'C' – Levels 1-4
- 'D' – Levels 5-8
- 'M' – Level 9

For example, if the current assembly architecture level is 5 and the label field contains DOIO, the architecture specific function name will actually be DOIOC.

ICALL DOIO is equivalent to
CALL DOIO5

IFUN

Source file: maclib/IFUN.mac

Macro Format:

```
label    IFUN  [FP=[YES|NO]
               [,AFP=[YES|NO]]
```

IFUN macro initiates a subroutine using the SATK standardized calling conventions performing input/output architecture specific functions by the current assembly architecture level.

Assembly Considerations:

- The same as for the FUNCTION macro.

Execution Considerations:

- The same as for the FUNCTION macro.

Label Field Usage:

The label field is required. It establishes the entry point of the function and is used directly by the ICALL macro to call the function or indirectly by the CALLR macro by previously loading a register with the label, suffixed by its input/output architecture specific suffix.

Positional Parameters: None

Keyword Parameters:

- The same as for the FUNCTION macro.

Programming Note:

Input/output architecture function suffixes are assigned by architecture level:

- 'C' – Levels 1-4
- 'D' – Levels 5-8
- 'M' – Level 9

For example, if the current assembly architecture level is 5 and the label field contains DOIO, the architecture specific function name will actually be DOIOC.

DOIO IFUN is equivalent to
DOIOC FUNCTION

LOCAL

Source file: maclib/LOCAL.mac

Macro Format:

```
LOCAL
```

The LOCAL macro allows the local stack frame of the following function, initiated by a following FUNCTION macro, to define fields used by the function within its stack frame.

The current control section is interrupted by positioning the assembly at the end of the standard fields in the stack frame DSECT, STKF in a 32-bit register environment or STKG in a 64-bit register environment.

The local fields are defined by following the LOCAL macro with one or more assembler DS directives.

Local stack frame usage is automatically terminated by a following FUNCTION macro.

Assembly Considerations:

- A preceding FRAME macro is required to establish the standard stack frame usage.
- A following FUNCTION macro is required to complete the definition of the local stack frame usage.

Execution Considerations: None

Label Field Usage: Ignored

Positional Parameters: None

Keyword Parameters: None

RETURN

Source file: maclib/RETURN.mac

Macro Format:

```
[label]   RETURN [ret]
```

The RETURN macro ends the function definition initiated by the preceding FUNCTION macro and provides the function epilog instruction sequence that:

- provides the caller, if requested, with a return value in general register 2,
- restores the caller's state and
- passes control back to the calling function following the caller's CALL or CALLR macro that initiated the function call.

If the function is returning any other values in register 3, 4, or 5, the returned values must be placed in the register before the RETURN macro is called.

The registers restored to the caller are controlled by the registers saved by the preceding FUNCTION macro (which is in turn influenced by the current architecture level and the assembly's FRAME macro parameters).

This table identifies the actual registers restored before returning to the caller:

Level	Type	FUNCTION FP=	FUNCTION AFP=	Restored
1-9	GR	--	--	R6-R15
1-9	FP	NO	NO	none
1-9	FP	YES	NO	F4, F6
7-9	FP	YES	YES	F8-F15

Assembly Considerations:

- Must be preceded by a FUNCTION macro.
- Any assembly or machine statements may follow.

Execution Considerations:

- Because only one RETURN macro is allowed per function, any need to provide an early termination of the function requires the function to branch to the function's RETURN macro via it label.

Label Field Usage:

- The label field is associated with the first instruction created by the `RETURN` macro and may be used to pass control to it through a branch type instruction.

Positional Parameters: None

Keyword Parameters: None

SCALL

Source file: maclib/SCALL.mac

Macro Format:

```
[label]   SCALL func
                [,INLINE=[A|J]]
```

The SCALL macro transfers program control to the a function performing sharable functions due to the same register size.

Assembly Considerations:

- The same as for the CALL macro.

Execution Considerations:

- The same as for the CALL macro.

Label Field Usage:

- The same as for the CALL macro.

Positional Parameters:

1. The func positional parameter identifies the architecture specific function being called. The parameter must identify a label that was established by a SFUN macro within the assembly.

Keyword Parameters:

- The same as for the CALL macro.

Programming Note:

Register size function suffixes are assigned by architecture level:

- 'F' – Levels 1-8
- 'G' – Level 9

For example, if the current assembly architecture level is 5 and the label field contains FORMAT, the architecture specific function name will actually be FORMATF.

SCALL FORMAT is equivalent to

```
CALL  FORMATF
```

SFUN

Source file: maclib/SFUN.mac

Macro Format:

```
label    SFUN  [FP=[YES|NO]
               [,AFP=[YES|NO]]
```

SFUN macro initiates a subroutine using the SATK standardized calling conventions performing sharable functionality due to register size by the current assembly architecture level.

Assembly Considerations:

- The same as for the FUNCTION macro.

Execution Considerations:

- The same as for the FUNCTION macro.

Label Field Usage:

The label field is required. It establishes the entry point of the function and is used directly by the SCALL macro to call the function or indirectly by the CALLR macro by previously loading a register with the label, suffixed by its input/output architecture specific suffix.

Positional Parameters: None

Keyword Parameters:

- The same as for the FUNCTION macro.

Programming Note:

Register size function suffixes are assigned by architecture level:

- 'F' – Levels 1-8
- 'G' – Level 9

For example, if the current assembly architecture level is 5 and the label field contains FORMAT, the architecture specific function name will actually be FORMATF.

```
FORMAT   SFUN is equivalent to
FORMATF  FUNCTION
```

STACK

Source file: maclib/STACK.mac

Macro Format:

```
label     STACK size
```

The stack used by the func category macros grows from the highest memory address of the stack (the bottom of the stack) to lower addresses. The point at which the STACK macro is used identifies the top of the stack (its lowest address). For the purposes of establishing the stack neither location is useful because the stack frame pointer must point to the first stack frame that resides at the bottom of the stack.

The required label is equated to the location of the first stack frame given a stack whose top address is the current location within the current control section, aligned to the next double word and size, rounded down to whole double words as the bottom of the stack. The stack defined by the STACK macro consumes no space in its control section but is positioned relative to the control section.

The first stack frame supports no local fields, being the stack frame used to preserve the state of the main program sequence. itself not being a function.

Assembly Considerations:

- The STACK macro requires a preceding FRAME macro within the assembly.

Execution Considerations: None

Label Field Usage:

- The label field associated with the location of the first stack frame is required.

Positional Parameters:

1. The required size positional parameter specifies the size of the stack and must be a self-defining term.

Keyword Parameters: None

STKINIT

Source file: maclib/STKINIT.mac

Macro Format:

```
[label]    STKINIT bottom
                  [,LOAD=[YES|NO]]
```

The STKINIT macro is used to initialize the stack frame pointer register, general register 15, with the address of the first stack frame at the bottom of the stack. The value of the bottom positional parameter is used to establish the content of the stack frame pointer.

If the bottom parameter is a register enclosed in parenthesis, the stack frame pointer is loaded from the contents of the specified register. It the bottom parameter is a label its treatment is determined by the LOAD keyword parameter.

Assembly Considerations:

- A preceding FRAME macro is required before invoking the STKINIT macro.
- To use the label created by the STACK macro, code:

```
STKINIT label,LOAD=YES
```

Execution Considerations:

- General register 15 is assumed to be the current stack frame pointer when a function is called, however its content is established (or not). If a meaningful location has not been established in general register 15, unpredictable results will likely occur.

Label Field Usage:

- The label field, if supplied, identifies the first instruction of the macro expansion.

Positional Parameters:

1. The bottom parameter is required. It may be a register enclosed in parentheses or a label.

Keyword Parameters:

Keyword	Default	Description
LOAD	NO	Specify: • NO to load the address constant identified by the bottom parameter label into the

Keyword	Default	Description
		stack frame pointer or • `YES` to create an inline address constant of the `bottom` parameter label and load it into the stack frame pointer. Ignored if the `bottom` parameter is a register.

HOWTO use Functions

The use of a program stack and associated function calls proves to be a very useful programming technique. SATK has adapted, with a few adjustments, the Executable and Linking Format (ELF) function calling conventions with a few adjustments for s390 and extensions for other CPU's.

See the “Appendix A – SATK Functions vs. ELF ABI” for details on how SATK functions differ from ELF ABI functions.

Why Functions?

Any branching instruction interrupts the default sequential instruction execution of a CPU. This is true for subroutine branching instructions. When using such instructions, the only state the subroutine must preserve is the location to which control must be returned when the subroutine has completed. This seems simple. And it is. However, for the assembly language programmer what becomes more complex is ensuring critical caller state, particularly in registers, is not lost while execution is occurring within the subroutine. If multiple subroutines are involved, or a subroutine needs to call another, the complexity starts to become unmanageable for the assembler programmer.

The natural recourse for this challenge is to establish a set of conventions observed by all subroutines and a reusable strategy for the preservation of program state during the call to a subroutine. Legacy mainframe programs use linked register save areas. The programmer has the responsibility of supplying the save area or areas if multiple calling depths are required. They were typically provided as a portion of the program, one save area per subroutine. Unless dynamic allocation of the save area is available, subroutine recursion is not possible. A lot is left up to the programmer to manage in this system.

The set of conventions and macros provided by this macro category alleviates nearly all of the drudgery required for generalized subroutine linkage supporting recursion, and, if functions are coded correctly, re-entrant usage. These conventions differ quite significantly from those used by legacy programs. Rather than a linked list of save areas, the program state, regardless of the number of subroutine calls, is preserved in the program stack. The program stack provides the simplest form of dynamic memory allocation for register save areas. It replaces the linked list of register save areas in legacy programs. Each subroutine gets a portion of the stack, a stack frame, for preserving its own state. The subroutines play by a set of rules that preserve the state and manage stack frames within the stack.

Program state requiring preservation is variable data (as opposed to constant data) that must

not be altered while a subroutine is executing on behalf of the program. Such variable data can reside in two places. One is memory used by the calling portion of the program. The second is resident within CPU registers at the point the subroutine is actually called. For the preservation of variable data in CPU registers, there is only one place where the data can be preserved, namely memory. The stack frame supports the preservation of both. The first form of variable data can reside in the stack frame itself while the subroutine is running as local stack data. The second form, CPU registers, must be transferred to storage, also in the stack frame, to preserve it, and returned to the registers by the called subroutine from the stack frame thereby preserving it. Once the subroutine's caller's state has been preserved, the subroutine is free to use CPU registers and store its own local data in its own stack frame. Somewhere in this process must a new stack frame become available for use of a called subroutine. For such interactions to occur, the program and all subroutines must support the same set of conventions, particularly with regard to CPU registers that participate in the process of state preservation and how each stack frame is structured.

The set of conventions for the purpose of simplifying the use of subroutines are collectively described here as “functions.” Functions, as implemented in the SATK solve much of the programming challenge in the use of subroutines for the assembly language programmer (and compiler developers too). It should be obvious that it is very desirable for these conventions' implementation be as efficient as possible. The conventions will be used in numerous programs and any inefficiencies will be magnified through this usage. The point at which a function call is made, the entering of the called function (prolog) and the point of return (epilog) by the called function become the critical points where the conventions are managed. It is within these key points that the stack is managed. The CPU instructions, the registers and roles within the CPU will drive the conventions.

The following SATK macros are provided for the support of functions.

Definition	Creation	Init/Term	Process
FRAME: STKF STKG	STACK LOCAL FUNCTION AFUN IFUN SFUN	Init: STKINIT	M: ACALL M: CALL M: CALLR M: ICALL M: RETURN M: SCALL

How the Stack Works

The SATK function stack is an area of memory in which function state is managed. Each caller of a function has an element on the stack, its stack frame, containing the caller's state. Each called function has its own element on the stack for its state. In the case of a function, the stack frame may optionally provide space in memory for use by the function itself.

The stack and its stack frames are integral to SATK implemented functions. The stack is a last-in/first-out (LIFO), or push down, stack. As stack frames are added to the stack, the stack grows to **lower** memory addresses. As stack frames are removed from the stack, it shrinks to **higher** memory addresses. The top stack frame, the most recent “last-in” stack frame, is located by the stack frame pointer (SFP). The SFP always points to the most recently called function's stack frame. Or, in the case of the main program, the main program's stack frame allowing it to call functions.

The stack uses a general register, 15, as the pointer to the current stack frame in use. As each function is called a new stack frame is added to the stack by placing it at a lower memory address, decreasing the value in general register 15 by the length of the stack frame being added. The reverse happens when a stack frame is removed. The value in general register 15 is increased by the size of the stack frame being removed, moving the stack frame pointer to the previous stack frame on the stack. The SFP always points to a double word boundary.

This diagram illustrates the structure of a stack. The main program has called function A. Function A uses some local fields within its stack frame. In turn function A has called function B. Function B is currently executing, so the SFP points to its stack frame. The stack frame used by Function A, because of the local fields, is larger the other stack frames. Stack frames vary in size, but are never shorter than the size required by the common fields..

UNUSED	**Function B**	**Function A**	**Main Program**
	Stack Frame 2	**Stack Frame 1**	**Stack Frame 0**
	`Common Fields`	`Common Fields` `Local Flds`	`Common Fields`
End SFP (R15)-->	Top		*Bottom* Start

Each caller of a function must have a stack frame on the stack. So the *Bottom* entry must exist before any function is called by the program. The `STACK` macro establishes the location of the *Bottom* entry by calculating where a stack would end (the Start+1 location) relative to the end of the program and then reserving space for the *Bottom* entry's common fields and giving it a label. `STKINIT` initializes the stack with the bottom entry by loading the *Bottom*

entry's stack frame address into the SFP (R15). This usage of STKINIT is only one way the macro can establish the location of the *Bottom* stack entry. The content of the *Bottom* stack frame is unknown at this point. However, after the program starts to call functions it will contain the main program's state during its function calls.

No mechanism at present exists within the SATK function implementation for detection of stack underflow or overflow. The program can provide overflow detection by examining the SFP following the FUNCTION macro. If the SFP contains a value less than *End*, stack overflow or SFP corruption has occurred. Stack underflow is not detectable. It requires examination of the SFP following the removal of the returning function but before the caller's stack contents are accessed and control returns to the caller. These two actions occur within the RETURN macro and would require an enhancement to it.

Each function is composed of three parts:

- a prolog used when control is passed to the function by its caller,
- the function body that does the work of function, and
- an epilog that passes control back to the function caller.

The function prolog and epilog is where caller state and stack management occurs.

Calling a Function

The CALL or CALLR macro initiates the transfer of control to a called function. In the process of doing that, the called function's base register is established in general register 13, and the return point is provided to the called function in general register 14.

Regardless of the architecture level in which the macros are used, the same functionality is provided. The architecture level may dictate how the functions are performed, meaning which instructions are used, but the results are the same.

Register Usage

The following table describes the register conventions of SATK functions. Volatile registers are highlighted in yellow. General registers are prefixed with the letter 'R'. Floating point registers are prefixed with the letter 'F'. Access registers are prefixed with the letter 'A'. Control registers are prefixed with the letter 'C'. 'FPC' designates the Floating-Point-Control register. Vector registers are prefixed with the letter 'V'.

The 'Levels' column indicates in which architecture levels the convention applies. The term

“caller' means either the main program or a function that calls a function.

Table cells in RED indicate the caller must itself preserve these register contents in either registers that are preserved across function calls, in memory (the main program) or local stack frame fields (function body). No action on the part of the caller is required if the contents do no require preservation across a function call.

Table cells in YELLOW indicate the caller must itself preserve these register contents in either registers that are preserved across function calls, memory (the main program), or local stack frame fields (function body) if the conditions indicated in the Note are not met. No action on the part of the caller is required if the contents do no require preservation across a function call.

Table cells in BLUE are only saved in the stack frame if `FRAME PACK=NO` and the condition in a note, when present, is met.

Table cells in GREEN are always saved in the stack frame and the condition in a note, when present, is met.

Level	Register(s)	Caller Before Function Call	At Function Body Entry	Upon Return to Caller
1-9	R0	Local usage	Unpredictable and available	unpredictable
1-9	R0, R1	Local usage	Unpredictable and available	unpredictable
1-9	R2-R5	Function arguments or local usage	Function arguments or unpredictable	Returned values or unpredictable
1-9	R6	Function argument	Function argument	Caller's value before call
1-9	R7-R12	Local usage	Available to function	Caller's value before call
1-9	R13	Local usage	Function base register	Caller's value before call
1-9	R14	Local usage	Caller's return location	Caller's return location
1-9	R15	Caller's SFP	Function's SFP	Caller's SFP
1-6	F0, F2 Note 3	Function arguments	Function arguments	Returned values or unpredictable
1-6	F4, F6 Note 3	Local usage	Available to function	Caller's value before call. Note 1
7-9	F0, F2, F4, F6 Note 4	Function arguments	Function arguments	Returned values or unpredictable.
7-9	F1, F3, F5, F7, F8-F15 Note 4	Local usage	Available to function	Caller's value before call. Note 2
7-9	FPC	Local usage	Available to function	Unpredictable
6-9	A0-A15	Local usage	Available to function	Unpredictable

Level	Register(s)	Caller Before Function Call	At Function Body Entry	Upon Return to Caller
2-9	C0-C15	Global usage	Available to function	Global state possibly altered
9	V0-V31	Local Usage	Available to function	Unpredictable

Note 1: The caller's value before the call provided `FRAME FP=YES` and `FUNCTION FP=YES`, otherwise unpredictable.

Note 2: The caller's value before the call provided `FRAME FP=YES,AFP=YES` and `FUNCTION FP=YES,AFP=YES`, otherwise unpredictable.

Note 3: Requires `FRAME FP=YES` and `FUNCTION FP=YES` for the registers to be saved on the stack, otherwise the registers are not saved.

Note 4: Requires `FRAME FP=YES,AFP=YES` and `FUNCTION FP=YES,AFP=YES` for the registers to be saved on the stack, otherwise the registers are not saved.

Function Prolog

The prolog is generated by the `FUNCTION` macro. Its first duty is to preserve the **caller's** register contents that are non-volatile in the **caller's** stack frame. The SFP contains the caller's stack frame location at the point control is passed to the function.

Next the prolog must “push” the called function's stack frame onto the stack. It does this by decremented the SFP by the size of the function's stack frame. The `FUNCTION` macro knows this because of information managed by the set of macro global symbolic variables used by this category of macros. The `LOCAL` macro manipulates this information informing the `FUNCTION` macro to allow space for local usage. The size is determined by how the `FUNCTION` macro terminates the stack frame DSECT containing the local fields defined following the preceding `LOCAL` macro. The size is always rounded up to full double words ensuring the SFP points to a double word boundary.

Function Body

As control passes out of the `FUNCTION` macro, the function body begins execution. The function body must analyze any function arguments passed to it by the caller.

If the function body itself calls a macro, even itself, if it needs a function argument preserved following the function call, it must preserve the argument either in a non-volatile register or in a local stack frame field.

If the function returns any values to the caller, they must be placed in the register established by the function's usage for its return. Only registers reserved for that purpose may be used.

If more arguments are required than are allowed by the registers reserved for argument passing, a DSECT and an area in memory can be used by the caller to pass them as a structure. The called function dictates the details of the structure. When this technique is used, a single argument, the address of the structure, can be passed to the called function. The structure may be passed as local fields within the calling function's stack frame.

The function body ends with the passing of control to the function's RETURN macro.

Function Epilog

The epilog is generated by the RETURN macro. It does the reverse of what the prolog does. It “pops” the function's stack frame from the stack by incrementing the SFP pointer by the length of the current function's stack frame. The RETURN macro knows the length by the information supplied to it by the FUNCTION macro. The SFP then points to the function's caller's stack frame. The epilog can then extract and restore the non-volatile registers of the caller to their values preceding the call to the function. Control is returned to the caller at the point following the call to the function at the address provided to the function in general the caller's register 14.

Because of the management of func macro category state, only one RETURN macro may exist within the function definition.

The Stack Frame

The stack frame consists of three areas, only one of which is required, the second:

1. A back pointer to the previous stack frame and accompanying language processor reserved field, both of which are either present or absent,
2. Required register save area and
3. Optional function local fields.

The first two areas are common to all stack frames (meaning all stack frames would reserve space for them when used, the first being optional). All functions will expect the same stack frame structure. The third area is local to a specific function and varies when used.

Function local fields are specific to a given function's use of the stack. For each function that uses local stack fields, the function definition must be preceded by a LOCAL macro and one or

more DS directives defining its local fields.

For many modern systems, the back pointer is not required for the support of functions. By default SATK does not include the back pointer or language processor reserved field in the stack frame structure, although it can be forced. Why one would want to do this when it is not needed and compatibility is not an issue is unclear. But these two fields can be forced.

The structure of the register save area is required and must be the same for all functions in the ASMA assembled bare metal program. The FRAME macro defines this format by creating a DSECT for the stack frame as discussed in the following “FRAME” section. Due to the difference in register sizes, the register save areas and other fields will differ in size between 64-bit and 32-bit register CPU's.

Before the stack can be established, the structure of the stack frame must be defined by placing the FRAME macro early in the assembly, but following the ARCHLVL or ARCHIND macros.

FRAME

The format of the stack frame must be defined before any functions may be called. The FRAME macro creates a DSECT, STKF in a 32-bit register architecture or STKG in a 64-bit register architecture. The FRAME macro may be called once in either of these contexts as implied by the current architecture level. Once called and having defined the stack frame format, it can not be changed for the entire program.

The assignment of volatile and non-volatile registers allows use of a single STORE MULTIPLE (STM or STMG) and LOAD MULTIPLE (LM or LMG) instruction when preserving and restoring, respectively, general register content. This is not the case for floating point registers. Each floating point register requires its own individual STORE (STD) and LOAD (LD) instruction.

Technically, only caller state that must be preserved requires storing of the register content into the stack frame. Most bare-metal programs do not require use of floating point registers. Usually the back pointer and compiler reserved fields are not needed. These considerations allow optional sizes for the stack frame itself and the number of instructions required to preserve a function caller's state.

Four parameters of the FRAME macro allow the **global** setting of the state that may preserved by functions defined in the program. If space within the stack frame is not defined then the corresponding state may not be saved by any function.

- BACKPTR controls whether space for a back pointer and compiler reserved field are reserved in the stack frame.
- FP controls whether space for floating point registers, specifically the four universally available floating point registers is reserved in the stack frame
- AFP reserves additional space for the additional 12 floating point registers available on some architectures.
- PACK removes reserved space from the stack frame for volatile parameter registers.

General Registers

Generally all parameter registers and registers required to preserve the calling function's state are saved in the stack. SATK by default only provides space for the general registers. Space for the volatile general registers is removed from the stack by specifying PACK=YES. Note that PACK=YES influences the space reserved for floating point parameter registers as well if space is reserved for them.

If PACK=YES is used, space for ten general registers, R6-R15 is available. If PACK=NO, space is reserved for fourteen registers, R2-R15. The registers saved in the stack frame matches the save area. Regardless of the value of the PACK parameter, only registers R6-R15 are restored.

Floating Point Registers

The need to manage floating point registers is an exceptional case for bare-metal programs. To provide space in the stack frame allowing saving of the universally available four floating point registers, specify the parameter FP=YES. For architectures that support the additional twelve floating point registers, space is provided by including the parameter AFP=YES. Both the FP and AFP parameters default to NO. FP=YES is required for AFP=YES to reserve space.

To disable the saving of floating point volatile registers, specify the parameter PACK=YES. This parameter removes space in the frame stack definition for floating point parameter registers.

The following table summarizes the floating point registers for which space is reserved. PACK=NO, FP=NO and AFP=NO are the default parameter settings.

For architecture levels 1-6 and 7-9 when AFP=NO.

PACK	FP	AFP	FP Registers	Reserves Space
NO	NO	NO	0	
NO	YES	NO	4	F0, F2, F4, F6
YES	NO	NO	0	
YES	YES	NO	2	F4, F6

For architecture levels 7-9 when FP=YES and AFP=YES.

PACK	FP	AFP	FP Registers	Reserves Space
NO	YES	YES	16	F0-F15
YES	YES	YES	12	F1, F3, F5, F7-F15

Whether any floating point registers are actually saved or restored in the reserved space is controlled by the corresponding FUNCTION macro parameters FP and AFP. The FRAME macro only reserves space allowing the FUNCTION to use the space reserved when its parameters direct it to do so.

Back Pointer and Compiler Reserved Field

To provide space for the back pointer and the compiler reserved field, use the BACKPTR=YES parameter setting. If space is allocated for the back pointer, it will automatically be saved into the stack frame during a function call. By default, BACKPTR=NO reserves no space for either the back pointer or the compiler reserved field. SATK does not utilize the compiler reserved field.

LOCAL

Usage by a function of the stack frame is supported by reserving additional space in the stack frame for the function's fields. The LOCAL and FUNCTION macros cooperate to achieve this result. The LOCAL macro initiates a scratch-pad area on the local function call stack. Initially the area is not initialized and all content is unpredictable upon entry to the function.

The LOCAL macro, coded before a function definition, is followed by DS assembler directives defining the use of the scratchpad area. The DS directives define and reserve space for the following function's usage of the stack frame. The DS assembler directives define the individual fields used by the function. The last DS assembler directive must be followed by the FUNCTION macro to both complete the stack frame definition for the function and define the

function's prolog logic.

Any use of DC assembler directives is the same as any other use of the DC assembler directive within a DSECT. The DC is treated as a DS assembler directive and constant information is discarded. A DC directive within the scratch pad area does not initialize the area. Explicit instructions moving or storing into the scratch pad area are required to place function values into it.

The LOCAL macro remembers the current control section and then makes the stack frame DSECT active, positioning the location counter following the space reserved by the FRAME macro. The subsequent FUNCTION macro ensures the stack remains on double words and returns to the control section remembered by the LOCAL macro. The stack frame size resulting from the additional space is used by the FUNCTION macro to establish the new stack frame pointer for the defined function in its prolog code.

The LOCAL macro must **not** be used between a FUNCTION and RETURN macro.

Initializing the Stack

The program stack is the core of function usage. The stack resides in memory. Before any functions can be called the location of the stack must be known. The bare-metal program must establish the program stack. The program stack grows downward in memory, starting at some address with stack frames added at lower memory locations. As frames are removed, the stack address returns to addresses higher in memory.

The STACK and STKINIT macros provide support for stack initialization.

STACK

The STACK macro defines an assembler label that represents the program's bottom of the stack location. This “bottom of the stack” address establishes the bottom stack frame. The STACK macro reserves space for preservation of the program state for the portion of the program initializing the stack. This allows the code that initializes the stack, using STKINIT, to then proceed to call functions dependent upon the stack. The initializing code, while being able to utilize functions, is itself not a function nor does it support local fields in the first stack frame. The first stack frame is strictly for the purpose of allowing the initializing code to call functions. The program stack frame format defines the size of this first stack frame.

STKINIT

The STKINIT macro performs run-time initialization of the stack. The stack must be initialized before any run-time function calls can be made. Unpredictable results will occur otherwise.

STKINIT accepts a run-time value in a register, a symbol of an address constant that defines the bottom of the stack or a symbol created by the STACK macro.

Function Definition

A SATK function is defined using two macros, the FUNCTION and RETURN macros, and optionally local stack frame usage as described in the previous section.

FUNCTION

The FUNCTION macro defines the entry point of the function and the prolog code.

The function prolog code:

- establishes the register conventions expected by the defined function,
- provides a base register of R13 for the function,
- saves the caller's state in the caller's stack frame,
- creates a new stack frame for use of the function, pointed to by R15, and
- saves the back pointer if the BACKPTR=YES was supplied on the FRAME macro.

The FUNCTION macro has only two parameters, the FP and AFP parameters. They define whether the function prolog and epilog code should save and restore the universal four floating point registers and the twelve additional registers if available. Save areas must have been reserved in the stack frame structure by the FRAME macro for these registers to actually be saved and restored. Specify FP=YES and optionally AFP=YES with the FUNCTION macro for the respective floating point registers to be saved by the FUNCTION prolog code, and restored by the RETURN epilog code.

Local frame usage by the function is defined previous to the FUNCTION macro that defines the function. The LOCAL macro must not be used between a FUNCTION and RETURN macro.

RETURN

The RETURN macro completes the function definition by supplying the function epilog code. It

must follow a FUNCTION macro. Only one RETURN macro may be used in a function definition, namely the one that terminates the function. Returning to a function's caller must use the RETURN macro. Multiple points from which a function may need to return must branch to the single RETURN macro to do so. Each FUNCTION macro must be paired with a corresponding RETURN macro.

The function epilog code:

- restores the caller's state and
- returns to the caller.

It is the responsibility of the function to return in R2-R5 any returned values before returning to the caller.

Using Functions

A function may be called either by

- its name (the CALL macro),
- a register containing the function's location (the CALLR macro), or
- an address constant pointing to the function being called (the CALLR macro).

CALL

The CALL macro uses a symbol to identify the function being called. For architectures that support it (levels 8 and 9) the symbol is used in a position relative instruction. All other architectures use an inline address constant.

The macro can be forced to use either approach to locating the called function by use of the INLINE parameter. INLINE=A forces use of an address constant. INLINE=J forces use of a positional relative long instruction.

If an address constant is used, it can be added to the self relocation environment by specifying RELO=YES.

CALLR

The CALLR macro accepts a register, in parenthesis, containing the address of the function being called or a symbol of an address constant pointing to the function being called.

If the parameter is omitted, the location of the function being called is assumed to be in

general register 1.

Structuring the Assembly

This section discusses how the assembly should be constructed when using the func macro category.

```
PROGRAM  START X'2000'
         USING *,13   Might as well use the same base as functions
* Initialize the application
         ARCHLVL      Establish the target architecture
* The func macros now know how to generate code for the target arch
         FRAME
* The STKF or STKG dummy section has been created.
          STKINIT MYSTACK,LOAD=YES    Initialize the frame stack ptr
* From this point forward R15 must not be used, except with extreme care
* Now functions can be defined and called
          LA     6,AFUNC
          CALLR  (6)     Call AFUNC using the register
          LTR    2,2     Check the return value for zero
          BZ     ...     Handle a zero return code

* Define the AFUNC function
AFUNC    FUNCTION
* Function body follows
         CALL   BFUNC
* Whatever is in general register 2 (from BFUNC) is the return value
         RETURN
* Function AFUNC definition ends

* Define the BFUNC function with local stack frame usage
         LOCAL
* At this point all statements are assembled into STKF or STKG
* dummy sections until the next FUNCTION macro occurs.
BFIELD   DS    F
BFUNC    FUNCTION
* FUNCTION sets the stack frame base register and using
         ST    2,BFIELD
         LTR   2,2
         BZ    BRETN
* Do some stuff setting the return value in R6...
         L     6,BFIELD
BRETN    RETURN 6
* Function BFUNC ends

* Set up the bottom of the function stack of 512 bytes
MYSTACK  STACK  512
         END
```

While it is not intuitively obvious, when functions are used, the main program will tend to adopt naturally the SATK function register conventions because it needs to use them when interacting with the functions it calls.

Appendix A – SATK Functions vs. ELF ABI

The Executable and Linking Format (ELF) specification describes how the format is used to create an object module that may be linked with other such modules and the resulting executable file that results from the linkage editing process. The ELF is used by Linux and other operating systems. As a by-product of its use by Linux, the ELF has been adapted for use on a large number of hardware platforms, including mainframe systems.

Each hardware platform that is supported with the general ELF specification has an Application Binary Interface (ABI) supplement to the general ELF. Inherent in the name are really two interfaces:

- the “binary interface” and
- the “application interface”.

The “binary interface” focuses primarily on the usage of the file formats by a specific processor. Sections within the ELF specifications identified as “processor specific” can be expected to have details supplied by the supplement. Tools which create these files or make use of them (for example an operating system program loader) are most interested in these aspects of the supplement and general specification.

The “application interface” is concerned with how programs can interact. Such interaction can take the form of memory based structures and the conventions involved in calling subroutines. There is naturally some interaction with the contents of the file formats. Language processor are influenced by the “application interface” aspect of the supplement.

The type of information involved in describing program interactions is:

- Register conventions,
- Subroutine prolog instruction sequences,
- Subroutine epilog instruction sequences, and
- Subroutine calling sequences.

The supplements have a strong orientation towards the C programming language. This is not surprising considering the ELF originated with UNIX System V and continues to incorporate that in its title.

The S/390 supplement applies to 32-bit Linux systems and the zSeries supplement applies to 64-bit mainframe systems.

- *S/390 ELF Application Binary Interface Supplement*, http://refspecs.linuxfoundation.org/ELF/zSeries/lzsabi0_s390/book1.html
- *zSeries ELF Application Binary Interface Supplement*, http://refspecs.linuxfoundation.org/ELF/zSeries/lzsabi0_zSeries.html
- *System V Application Binary Interface – DRAFT*, 10 June 2013, http://www.sco.com/developers/gabi/latest/contents.html

Because SATK ASMA assembler does not directly support the ELF standard object file format, the macros provided for use with ASMA are not actually ABI compliant. Use of similar macros in the files `ABI-lcls.S`, `ABI-s390.S`, `ABI-s390x.S` and `ABI.S` found in the `src` directory for use with the GNU assembler are ABI compliant. The GNU assembler, `as`, does actually support the ELF file format.

The following section provide additional details with regard to specific registers and how their SATK ASMA assembler function usage may differ from those identified in the S/390 or zSeries ELF ABI references.

Register Usage

The ELF ABI supplements describe register usage. Register content is described as “saved” or as being “volatile”. These terms are expressed from the point at which a function is called, and whether the content of a register can be expected to be the same before and after the call. Saved registers have their contents preserved across a function call. Volatile registers do not.

The supplements referenced above are oriented to a certain level of architecture that provides certain capabilities. The SATK attempts to provide support for all levels of the architecture, some of which may not offer those capabilities.

Assuming a register is used to point to a function's own stack frame, that register would need to be preserved across function calls.

On mainframe systems, establishing the return location for a subroutine call (which is in essence what a function is) requires a register. Whatever may be in it before the call to a function, it will obviously be different as a result of the call. So that register is volatile.

The ABI has its roots in the C programming language. This influences some aspects of the use of registers. For example, a C function can return one and only one value as a result of the function call. Some number of registers, platform dependent, are allowed to contain parameters of the call. For mainframe systems this is 4, one of which being the register that

contains the value returned by the function. These registers are described as volatile. The return value register is obvious. Whatever was in the register before the call will be replaced by what the called function returns. The reason the other parameter registers are described as volatile is less obvious. Realizing that a called function can itself call a function using these same parameter registers for its call makes it clear why they are likely to be volatile.

Mainframe systems utilize different registers for floating point values than those for binary integer data. Floating point function parameters and return values are defined in terms of the floating point registers with similar consideration for preservation or volatility.

If any registers are altered as a result of the function call process itself in either the prolog or epilog portions of the function, those must also be considered volatile.

Ultimately it is the responsibility of the **called** function to preserve those registers designated as not being volatile. It is the responsibility of the **calling** function to preserve across function calls those values of volatile registers, usually saving the value within its portion of the stack frame, before a function call.

R2-R5

The ELF ABI supplements restrict a function to a single return value in R2. The SATK extends the use of registers for return values to any of the additional parameter passing registers.

R12

The ELF ABI supplements use R12 for the address of the global offset table, aka the GOT. The GOT is specific to the ELF file format. Accessing global addresses is straight forward in assembly. Simply defining an address constant for the location suffices. SATK extends the use of R12 to the local function.

If the bare-metal program has a structure that forms the basis of its global context, usage of R12 for its address would be consistent with the ELF ABI usage of R12.

R13

The ELF ABI supplements define the use of R13 as “Local variable, commonly used as Literal Pool pointer”. The architectures targeted by the ELF ABI supplements all have the ability to use the PSW instruction address for branch relative addressing. This eliminates the need of a base register required for branching within the function. Some architectures supported by SATK do not have this option. Branching within the function requires a base register in these cases. For this reason, the SATK establishes a base register for the entire function using R13.

R14

Establishing the return location for a subroutine call (which is in essence what a function is) requires a register. A function's return address is placed in R14. Whatever may be in it before the call to a function, it will obviously be different as a result of the call. So R14 is volatile.

Generally, R14 should be avoided. However, if a function must use R14, it must restore its value before using its `RETURN` macro.

R15

A register is used to point to a function's own stack frame, R15. While changing when function calls another function, from the viewpoint of the function caller, R15 register is preserved across function calls.

While the rules can be more flexible within the SATK, the one register that should be avoided by the program is R15 when a program stack is in use. It is managed by the SATK function macros. If a function must use R15, it must restore its value before using the `CALL` macro or its own `RETURN` macro.

A0, A1

The ELF ABI supplements restrict the use of A0 and both A0 and A1, depending upon the supplement, to system usage. The actual usage is in support of thread local storage. The address of the thread local storage is supplied in register A0 or both A0 and A1. SATK does not support the concept of thread local storage, so makes these volatile registers generally available to functions.

Appendix B – SATK Functions vs. Legacy Subroutines

Legacy subroutines typically utilized a linked list of register save areas. When a subroutine is called:

- a register save area, the
- subroutine entry point, the
- return location, and,
- optionally, parameters

are all passed to the subroutine. In this regard legacy subroutines are similar to SATK functions. However the details are different. This section is primarily focused on the differences. Some will find advantages and disadvantages between the two styles.

While no explicit support for legacy subroutines is provided by SATK, they can readily be implemented using ASMA and the information found here:

- I*BM® DOS Supervisor and I/O Macros*, GC24-5037-12, October 1973, pages 244-247.

Register Usage

Both styles reserve R13-R15 for use by the calling convention, although with different uses.

Only one register is “volatile” from the perspective of a subroutine caller, R13. All other registers can be preserved across subroutine calls.

Subroutine base register will usually be the subroutine's entry point, R15. This is similar to a function's base register, although it is R13.

Parameter Passing / Returning Results

Legacy subroutines provide two registers for passing parameters: R0, and R1. If more than two parameters are required, a parameter list pointed to by R1 is used to pass the parameters. The parameter list is a contiguous sequence of values, the last of which has bit 0 set to 1, indicating the end of the list.

Result values must be returned either by use of a parameter that identifies where the result is stored or by storing the result in the register save area so that the subroutine caller has access through the restored registers.

SATK functions provide five registers for the passing of parameters, R2-R6; and four registers

for the returning of results, R2-R5. The parameter list style could be used with SATK functions by placing the parameter list address in R2.

Register Save Areas vs. Stack Frame

The legacy subroutine save area provides some of the functionality of a function stack frame, namely register preservation and nested subroutine calls. The static nature of the save areas eliminates the ability to perform subroutine recursion, while function recursion including self recursion is supported by SATK functions.

It is certainly possible to enhance legacy subroutines with a stack like structure that would allow recursion.

The one thing not available with legacy subroutines is local storage. For re-entrant subroutines, local storage is required and some form of memory management is necessary for this somewhere. Enhancement of legacy subroutines to support that is of course possible. Use of an operating system that provides some of these services is helpful. But, bare-metal programs do not have an operating system.

After the various enhancements to legacy subroutines are provided to get the same capabilities offered by SATK functions, why not just use functions instead. In fact SATK originally took the path of legacy subroutines during the earliest GNU `as` period of development. This was abandoned for functions because of the limitations of legacy subroutines.

One drawback of legacy subroutines for bare-metal programs is the presence of the register save areas within the code. In the image file or list-directed IPL program produced by an assembler, each register save area actually consumes space. This increases the size and number of an IPL program or number of records required to load a program with a boot loader. SATK functions eliminate this concern by establishing the stack at run-time outside of the actual loaded program. No stack frames are part of the loaded program nor do they take up space in space constrained media IPL records.