Table of Contents

Notices

z/Architecture is a registered trademark of International Business Machines Corporation.

Introduction

The Stand-Alone Tool Kit (SATK) provides tools and source code assisting the development of stand-alone programs for mainframe real, virtual or emulated hardware environments. Initially SATK used the GNU `binutils` project's assembler, `as`, and linker, `ld`. With the addition of A Small Mainframe Assembler (ASMA) to SATK, GNU `binutils` support is currently stabilized, and all future efforts will be developed using ASMA.

This document describes various facilities provided by SATK using ASMA and their conventions and standards. These facilities fall into two broad categories:

- hardware interaction by stand-alone software and
- assists for stand-alone software development itself.

An examples of hardware interaction might be establishing interrupt service routines, performing input/output operations or defining assigned storage usage. Examples of the second could be creating tables and searching them or establishing function calling conventions.

All of the the software described in this manual resides in the `srcasm` directory of SATK.

The manual is organized with summary sections at the beginning followed by detail section focused on individual areas.

General Source Conventions

Files that end in a suffix of `.asm` contain software that generates program content at the point at which they are copied into the assembly. Files that end in a suffix of `.mac` contain macro definitions that only generate program content at the point at which a macro is invoked but not at the point at which the source file is copied into the assembly. All file names in the `srcasm` and its sub-directories will use lower case. This ensures no conflict between hosting platforms that have a file system that is case insensitive versus those that are.

The ASMA `ASMPATH` environment variable must include at least the `srcasm` directory. Copied file names may contain a sub-directory or the sub-directory may be added to the `ASMPATH` environment variable as the user selects.

Term Definitions

SATK uses certain terms to define how a component is utilized in the context of other instruction sequences.

Inline refers to macro generated instructions intended for usage within the current sequence of instructions.

Routine refers to a separate program sequence entered using instructions such as BRANCH AND LINK or BRANCH AND SAVE. Calling conventions are specific to the routine.

Function refers to a routine accessed via a set of SATK standardized calling conventions.

Interrupt Service Routine **(ISR)** refers to a separate program sequence entered via the acceptance by the CPU of an interrupt.

Program is any number of instructions executing in a bare-metal environment. This may be designed for a specific purpose or a more general purpose such as an operating system kernel. See the “SATK Use Cases” section for more information.

Summary Reference

For convenience, this section includes summary information useful for the experienced user.

Disabled Wait States

Stand-alone programs frequently communicate termination conditions using disabled-wait state program status words. In such a PSW, interruptions are disabled and the PSW places the processor into a wait state. Because the processor is disabled for interrupts, externally initiated functions, for example an Initial Program Load or restart are required to cause further instruction execution. For this reason they are chosen to communicate the termination condition. The PSW instruction address defines the information containing aspect of the termination state. Inspection of the current PSW within the environment is required to ascertain the address. This section documents the disabled wait states used by SATK.

SATK reserves all addresses from `X'0001'` through `X'FFFF'` for its use. The following table documents these codes. By convention, an address of all zeros means successful termination of the stand-alone program. Stand-alone programs are encouraged to use disabled-wait state codes for abnormal terminations starting at `X'10000'` by allowing the `DWAIT` macro in `satk.mac` `PGM` parameter to default to `'1'`.

PSW Address	Component	Description
0000	DWAITEND	Successful termination
0008	TRAP64	Restart interrupt occurred without a restart ISR
0018	TRAP64	External interrupt occurred without an external ISR
0020	TRAP64	Supervisor-call interrupt occurred without a supervisor-call ISR
0028	TRAP64	Program interrupt occurred without a program ISR
0030	TRAP64	Machine-check interrupt occurred without a machine-check ISR
0038	TRAP64	Input/output interrupt occurred without an input/output ISR
0120	TRAP128	z/Architecture® restart interrupt occurred without a restart ISR
0130	TRAP128	z/Architecture external interrupt occurred without an external ISR
0140	TRAP128	z/Architecture supervisor-call interrupt occurred without a supervisor-call ISR
0150	TRAP128	z/Architecture program interrupt occurred without a program ISR
0160	TRAP128	z/Architecture machine-check interrupt occurred without a machine-check ISR
0170	TRAP128	z/Architecture input-output interrupt occurred without an input/output ISR
0200	ANTR	Running architecture is unknown or invalid.
0201	ANTR	Program does not support use on a System/360
0202	ANTR	Program does not support use on System/370 in basic-control mode
0203	ANTR	Program does not support use on System/370 in extended-control mode

PSW Address	Component	Description
0204	ANTR	Program does not support use on Hercules System/380
0205	ANTR	Program does not support use on System 370 Extended Architecture system
0206	ANTR	Program does not support use on ESA/370 system
0207	ANTR	Program does not support use on native ESA/390 system
0208	ANTR	Program does not support use on z/Architecture system in ESA/390 mode
0209	ANTR	Program does not support direct entry in native z/Architecture mode
0400	HRCCMD	Hercules DIAGNOSE X'008' command failed
10BAD	DWAIT	Program abnormal termination due to an unspecified condition.

SRCASM Files

This section describes in general terms the contents of supplied ASMA source files.

File	SATK FEAT	Description
function.mac	FUN	Stack-based callable functions
hercules.mac	HRC	Macros tailored for use with Hercules emulator.
satk.mac		Assists for basic machine architecture and operations.
selfrelo.mac	**REL**	Program embedded self relocation
table.mac	TBL	Generate and work with sequential fixed size entry tables

Components

This section summarizes the various SATK components. Individual files and components should be consulted for details.

The same component name may appear more than once. For example, ORB is both a macro found in satk.mac and a DSECT created by the IOFMT macro.

Component	Related To	Type	Description
ACALL	Function.mac	macro	Calls a function sensitive to the current architecture level
AFUN	function.mac	macro	Defines an function sensitive to the current architecture level
ANTR	satk.mac	macro	Uses run-time detected architecture level to pass control to supporting portion of the program
APROB	satk.mac	macro	Run-time detection of current architecture level.
ARCHLVL	satk.mac	macro	Detects assembly-time architecture level based upon the current XMODE PSW setting.
ASAIPL	satk.mac	macro	Introduce into a created assigned storage region an IPL PSW for

Component	Related To	Type	Description
			program entry
ASALOAD	satk.mac	macro	Create an assigned storage region containing trap PSW's
ASAREA	satk.mac	macro	Defines assigned storage locations used by all architectures in the first storage page.
ASAZAREA	satk.mac	macro	Defines z/Architecture specific storage locations used in the second assigned storage page.
CALL	function.mac	macro	Call a function using an inline address
CALLR	function.mac	macro	Call a function based upon a register resident address
CCWFMT	struct.mac	macro	Defines format and field values of Channel Command Words
CCW0	IOFMT	dsect	Format-0 Channel Command Word definition
CCW1	IOFMT	dsect	Format-1 Channel Command Word definition
CPUWAIT	satk.mac	macro	Performs in-line synchronous wait for an I/O or external interruption.
CSW	IOFMT	dsect	Channel Status Word definition
DSECTS	satk.mac	macro	Unifies structure definition into a single macro for various structures
DWAIT	satk.mac	macro	Generates and optionally loads a formatted disabled wait PSW
DWAITEND	satk.mac	macro	Generates and optionally loads a normal termination disabled wait state PSW
ENADEV	satk.mac	macro	Enable a device preparing it for use.
ESA390	satk.mac	macro	Run-time change to ESA/390 architecture mode from z/Architecture.
FRAME	function.mac	macro	Define the function stack frame
HRCIPLP	hercules.mac	macro	Store Hercules IPL parameters in the first 4096 bytes of storage.
HRCCMD	hercules.mac	macro	Issue a Hercules command without a response.
ICALL	function.mac	macro	Calls a function sensitive to the current architecture's input/output system.
IFUN	function.mac	macro	Defines a function sharable with other architectures utilizing the same input/output system as the current architecture.
IOFMT	satk.mac	macro	Defines various input/output related structures
IOCB	satk.mac	macro	Creates the control block for a specific I/O operation by RAWIO.
IOCBDS	satk.mac	macro	Defines the IOCB structure used by RAWIO.
IOINIT	satk.mac	macro	Prepare the I/O system for use.
IOTRFR	satk.mac	macro	Calculates the number of bytes actually transferred during an I/O operation.
IRB	IOFMT	dsect	Interruption Response Block definition
LOCAL	function.mac	macro	Define the local usage of the stack frame by the following function.
ORB	satk.mac	macro	Assembles an Operation Request Block or a reserved bit mask
ORB	IOFMT	dsect	Operation Request Block definition
POINTER	satk.mac	macro	Assembles an architecture sensitive address constant

Component	Related To	Type	Description
PSWB	PSWFMT	dsect	64-bit basic PSW format definition
PSWE	PSWFMT	dsect	64-bit extended mode single or bimodal addressing format PSW definition
PSWZ	PSWFMT	dsect	128-bit trimodal addressing format PSW definition
PSWFMT	satk.mac	macro	Defines PSW formats based upon current architecture level
RAWAIT	satk.mac	macro	Perform an inline wait for an unsolicited device interruption
RAWIO	satk.mac	macro	Perform an inline low-level I/O operation with a device.
RETURN	function.mac	macro	Return control to a function's caller.
SCALL	function.mac	macro	Calls a function sensitive to the current architecture's CPU register size
SCHIB	IOFMT	dsect	Subchannel Information Block definition
SCSW	IOFMT	dsect	Subchannel Status Word definition
SFUN	function.mac	macro	Defines a function sharable with other architecture's utilizing the same register size.
SIGCPU	satk.mac	macro	Generic run-time signaling of the running or other CPU.
SSMGR	satk.mac	macro	Static memory manager algorithm
SSMGRB	satk.mac	macro	Static memory manager algorithm control fields
TRAPS	satk.mac	macro	Generates or enables trap PSW for unsupported interrupt service routines.
ZARCH	satk.mac	macro	Run-time change to z/Architecture mode from ESA/390.
ZEROH	satk.mac	macro	Set to 0 high-order bits of a 32- or 64-bit register.
ZEROL	satk.mac	macro	Set to 0 low-order bits of a 32- or 64-bit register
ZEROLH	satk.mac	macro	Set to 0 high-order bits starting at bit 32 of a 64-bit register.
ZEROLL	satk.mac	macro	Set to 0 low-order bits ending at bit 32 of a 64-bit register.

Architecture Levels

The symbolic variable `&ARCHLVL` defines the current architecture level in use by the program. `&ARCHLVL` is established by the `ARCHLVL` macro, in `satk.mac`, by examining the current `XMODE PSW` setting.

0 – architecture level not established or unknown,

1 – System/360 (S/360),

2 – System/370 (S/370) in basic-control mode

3 – System/370 in extended-control mode

4 – System/380 (S/380, a Hercules unique architecture)

5 – System/370 Extended Architecture (370-XA)

6 – Enterprise System Architecture/370 (ESA/370)

7 – Enterprise System Architecture/390 (ESA/390)

8 – Enterprise System Architecture/390 on a z/Architecture system, and

9 – z/Architecture

Key new capabilities offered by the various architecture levels. The columns with an 'H' indicate Hercules unique offerings. This list is not exhaustive but representative or those features of interest to SATK applications.

1	2	3	4	5	6	7	8	9	Capability
X	X	X	H						Parallel Channels
X	X	X	H	X	X	X	X	X	12-bit displacements - 4K
X	X	X	H	X	X	X	X	X	Format-0 CCW
	X	X	H	X	X	X	X	X	BRANCH AND SAVE
			H	X	X	X	X	X	31-bit addressing
				X	X	X	X	X	Channel Subsystem
				X	X	X	X	X	Format-1 CCW
	H	H	H			X	X	X	Relative instructions - +/-64K (BRC, J, etc.)
	H	H	H			X	X	X	Immediate instruction (CHI, AHI, etc.)
						X	X	X	12 additional floating point registers
						X	X	X	Floating-point control register
						X	X	X	Binary floating point
							X	X	Addressing mode instructions (SAMxx, TAM)
								X	Transport Mode (TCW)
								X	64-bit addressing
								X	64-bit registers
								X	Relative long instructions +/- 2G (LARL, JLU, etc.)
								X	Extended immediate (CFI, etc.)
								X	20-bit displacements - 1M (LAY, AY, ALY, etc.)
								X	Decimal floating point

Operation Synonyms

Operation synonyms are defined by the `ARCHIND` macro in the file `satk.mac`. The native instruction mnemonic to which the synonym is assigned by architecture level is provided in the following table. `'--'` indicates no form of the instruction is supported by the architecture.

Synonym	0	1	2-6	7, 8	9
$AHI	--	--	--	AHI	AGHI
$AL	AL	AL	AL	AL	ALG
$ALR	ALR	ALR	ALR	ALR	ALGR
$B	B	B	B	J	J
$BAS	BAS	BAL	BAS	BAS	BAS
$BASR	BASR	BALR	BASR	BASR	BASR
$BC	BC	BC	BC	BRC	BRC
$BCTR	BCTR	BCTR	BCTR	BCTR	BCTGR
$BE	BE	BE	BE	JE	JE
$BH	BH	BH	BH	JH	JH
$BL	BL	BL	BL	JL	JL
$BM	BM	BM	BM	JJM	JM
$BNE	BNE	BNE	BNE	JNE	JNE
$BNH	BNH	BNH	BNH	JNH	JNH
$BNL	BNL	BNL	BNL	JNL	JNL
$BNM	BNM	BNM	BNM	JNM	JNM
$BNO	BNO	BNO	BNO	JNO	JNO
$BNP	BNP	BNP	BNP	JNP	JNP
$BNZ	BNZ	BNZ	BNZ	JNZ	JNZ
$BO	BO	BO	BO	JO	JO
$BP	BP	BP	BP	JP	JP
$BXLE	BXLE	BXLE	BXLE	JXLE	JXLEG
$BZ	BZ	BZ	BZ	JZ	JZ
$CH	CH	CH	CH	CH	CGH
$CHI	--	--	--	CHI	CGHI
$L	L	L	L	L	LG
$LH	LH	LH	LH	LH	LGH

Synonym	0	1	2-6	7, 8	9
$LPSW	LPSW	LPSW	LPSW	LPSW	LPSWE
$LR	LR	LR	LR	LR	LGR
$LTR	LTR	LTR	LTR	LTR	LTGR
$NR	NR	NR	NR	NR	NGR
$SL	SL	SL	SL	SL	SLG
$SLR	SLR	SLR	SLR	SLR	SLGR
$SR	SR	SR	SR	SR	SGR
$ST	ST	ST	ST	ST	STG
$STM	STM	STM	STM	STM	STMG
$X	X	X	X	X	XG

Assembler Source Structure

SATK is intended to make bare-metal programming easier, not harder. SATK provides a number of coded and tested source modules for building stand-alone programs with ASMA. These modules are in the `srcasm` directory. Any ASMA program using these source modules must include the `srcasm` directory in the `ASMPATH` environment variable directory search order list before executing the assembler.

Core capabilities are provided by the `satk.mac` source module. All users of ASMA source files will need to use the assembler `COPY` directive to include this file early in the program. Other capabilities must be manually included by use of a `COPY` directive for the needed file. This manual describes the capabilities offered by each file separately, discussing how the various components are for the file are used. Most separately included files have dependencies on the `satk.mac` source module, usually for architecture sensitive code generation or structure definitions. If files in `srcasm` are used, the `satk.mac` file must be included, unless otherwise indicated.

This manual is organized by file name with descriptions of each capability's design. The actual source file provides the ultimate definition of functions provided. Where the software or this document differ, this document is the basis for “working as designed”. Errors of course can exist in the code or this documentation. Either requires correction.

Always refer to the actual source module for detailed macro parameter descriptions.

SATK Use Cases

The relationships between various components can be quite complex in a generalized system. The support by SATK of multiple hardware architectures complicates these considerations. SATK recognizes two use-cases:

1. a stand-alone application performing one or more specific functions and
2. a control program providing resource management for one or more activities.

An application performs a single task. It supports use of the hardware in the way any application would, doing one piece of the task at a time. A control program is intended to manage the system resources for one or more application type tasks. The second use case has a much higher complexity. In both cases the tool kit only contributes to the end product. The support for each case may have elements in common with each other or entirely different ones. In some cases the same conceptual elements may be needed but implemented differently for the two environments.

An application typically does not require any complex interrupt handling. Although an ISR to provide information when a program interruption occurs might be desirable. It utilizes memory without consideration of other users, because of course there are none. All devices are dedicated to the use of the application. Examples of an application could be a boot loader

or program installer onto IPL media. A “Hello World” program would be another.

A control program **does** require true interrupt handling, activity management and perhaps memory segregation, protecting different activities from each other. Devices may be shared or dedicated to specific activities. An example of a control program might be a simple kernel or program debugger. In the early stages of the control program it might need to operate like an application until it is ready to function as a resource controller for its activities. The SATK does not attempt to provide an interface to a control program for its activities. That is the role of a control program itself. Nor does SATK make any attempt at integrating the pieces it supplies into a unified application. That is the role of the actual program.

The following table looks at various capabilities needed by an application as opposed to a control program. The table assumes that a control program starts out as an application that initializes the control program itself. The table is of course not exhaustive and capability needs could be argued either way. The intent is to drive towards a source structure that matches the two use cases under consideration. Rows describe a capability. The capabilities are placed generally in a spectrum (subject to some debate) from simplest to most complex.

The meaning of the colors is explained below.

Appl.	CP Init.	CP	Type	Capability
yes	yes	yes	system	Tailor for architecture
yes	yes	yes	system	Assigned storage location definitions used
likely	maybe	no	I/O	Perform I/O without ISR
maybe	yes	no	storage	Assign memory usage
maybe	maybe	no	system	Detect run-time environment
maybe	maybe	no	CPU	Use timers without ISR
maybe	maybe	unlikely	storage	Perform self relocation
maybe	yes	maybe	CPU	Initialize ISR's
maybe	maybe	maybe	system	Platform specific support (Hercules, or others)
maybe	likely	likely	program	Manage tables
maybe	maybe	likely	program	Use function calls
maybe	maybe	likely	program	Format text output
maybe	maybe	yes	I/O	Understand specific device characteristics
maybe	maybe	yes	I/O	Discover and identify devices
maybe	maybe	yes	CPU	Utilize interrupt service routines (ISRs)
maybe	maybe	yes	CPU	Dispatch interrupts within ISRs
no	no	yes	program	Manage resources for activities

Appl.	CP Init.	CP	Type	Capability
no	no	yes	program	One or more activities supported
no	no	yes	program	Activity services provided
no	no	yes	storage	Manage memory area allocation/release

From the table is is clear that the CP initialization does in fact model closely an application. There is also a clear demarcation in what an application or control program initialization component requires from that of an actual control program. The actual needs of a control program are specific to it. While the goals of SATK include a simple control program, it is out of scope for this manual focused on the contents and use of the `srcasm` directory. So nothing identified in red above can be found in that directory.

The components that may likely be required for an application are maintained in a single file, `satk.mac`. This file includes macros supporting everything in blue and green. It will normally be copied into the SATK application at its beginning. Use the `PRINT OFF` directive before and the `PRINT ON` directive to remove the files contents from the listing. Use the `NOPRINT` operand of the `PRINT` directive to remove the `PRINT` directives themselves from the listing.

The capabilities in yellow and orange are areas of overlap. The capabilities in yellow are likely of use to an application and those in orange are possibly needed by a control program. Capabilities in yellow or orange are made available in separate files that must by explicitly included into the program by means of the `COPY` directive.

Most components fall into one of four roles:

- Definition – assembles one or more DSECTs describing control block structures,
- Creation – assembles one or more control structures matching its definition,
- Initialization/Termination – Run-time environment creation or destruction, or
- Process – run-time processes supported by the environment.

The following table will be used to summarize the role individual components have in providing the SATK capability.

Definition	Creation	Init/Term	Process
`MACRO`: `DSECT`...	`MACRO` : `DSECT`	Init: `MACRO`... Term: `MACRO`...	M: `MACRO` R: `ROUTINE` F: `FUNCTION`

Supported Run-time Environment

The present state of development expects to execute in a single-CPU / single-thread

programming model. The state of the CPU constitutes the state of the “single-thread'. SATK does not implement any concept of a task or thread control structure. While the SATK macros as of yet do not support multiple threads or multiple CPU's, their is no constraint on a user from doing so in the user's own programs. The `SIGCPU` macro could provide the foundation for support of multiple CPU's, but at present is only used by SATK to change architecture mode.

The more complex the bare-metal program becomes, the more it moves from use case 1 towards use case 2. Presently SATK supplied assembler source targets use case 1 in the single thread, single CPU model. Inherent in the above chart is this natural progression of complexity. A long-term objective of SATK is the creation of a small control program / micro-kernel built upon the foundation macros supplied by SATK. Exactly where one use case ends and the other begins is yet to be decided. The second use case is likely to be a sub-project of its own within the SATK, having its own source directory.

Generally, it is expected that a bare-metal program will be built for a single architecture level. However, supporting more than one architecture with the same bare-metal program can be achieved.

Memory Usage

Use of SATK macros will usually result in a program of multiple regions:

- an IPL PSW region or
- an assigned storage area region (containing an IPL PSW) and
- the region containing the program.

Use of the ASMA `--image` output option is not recommended unless the program has been constructed to expect that option is in use. All of the other ASMA output options can accommodate a program of multiple regions. Use of the ASMA list-directed IPL option, `--gldipl`, is encouraged when using the Hercules emulator.

Local Modifications

The use of any of the source modules described in this manual is strictly optional. And if the code provided does not meet your needs you are free to modify it. It is recommended that modification be made outside of the SATK provided files in other directories, preserving the original source unchanged. Such directories must be included in the `ASMPATH` environment variable as is the `srcasm` directory to allow the files to be found by the ASMA `COPY` directive. To override SATK supplied source modules in their entirety, place the directory where the modified files exist, before the `srcasm` directory.

To override specific SATK supplied macros, place, the modified macros in a file residing in a directory other than `srcasm` with a name that does match any of the files in `srcasm`. Place this directory after the `srcasm` directory in `ASMPATH`. The reason for this is that a macro defined with the same name as a previous macro overrides its previous definition. This new

file could of course be placed in the `srcasm` directory itself, but doing so would mix local modifications with SATK, a practice which is discouraged.

To avoid any potential `ASMPATH` sequence issues, use a different directory with different file names and macro names unique to your modifications. This practice allows the directory containing local modifications to be placed anywhere in the `ASMPATH` search order. Other strategies are possible. The use of the `ASMPATH` environment variable is the enabler.

satk.mac

File `satk.mac` provides the following capabilities:

- Architecture Sensitivity – Supports code sensitive at assembly time (macros `ARCHLVL`, `ARCHIND`) or run-time (macros `APROB` and `ANTR`) to architecture levels.
- Define Structures - assemble various hardware and SATK related structure definition DSECTS (macros `ASAREA`, `ASAZAREA`, `DSECTS`, `IOCBDS`, `IOFMT`, `PSWFMT`).
- Change Architecture Mode and signal processors (macros `ESA390`, `SIGCPU`, and `ZARCH`).
- Program Termination - normally (macro `DWAITEND`) or abnormally (macro `DWAIT`) terminate a program.
- Flying without a Net - Waiting and other considerations for programs without formal interrupt service routines (macros `ASALOAD`, `CPUWAIT`, `TRAP64`, `TRAP128`, `TRAPS`).
- Performing I/O – In-line initiation and completion of low-level device I/O in either channel or channel subsystem environments (macros `ENADEV`, `IOCB`, `IOINIT`, `ORB`, `RAWIO`).
- Static Memory Allocation – run-time allocation of unassigned memory locations (macro `SMMGR` and `SMMGRB`).

Architecture Sensitivity

SATK is intended to facilitate coding for any of the architectural systems in the mainframe family. The foundation for this is the use of the Machine Specification Language (MSL) files that define a CPU instruction set and the format of its critical system structures, the Program Status Word (PSW) and Channel Command Word (CCW). The MSL specified PSW, by default, becomes the initial `XMODE` directive `PSW` specification. See the *ASMA* manual for details concerning the `XMODE` directive and initial CPU targeting by the ASMA command line.

Architecture sensitivity is supported by five macros:

- `ARCHLVL` – establishing the current assembly time architecture level
- `ARCHIND` – establishing operation synonyms for use in generic statements and
- `APROB` – establishing the run time architecture level.
- `ANTRY` – passes control based upon detected run-time architecture level
- `POINTER` – assemble an architecture sensitive address constant

Definition	Creation	Init/Term	Process
--	ARCHLVL ARCHIND	--	M: APROB M: ANTR M: POINTER

APROB

The APROB macro is one of the few macros not dependent upon architecture level. It is also not dependent upon the instruction set of the command-line selected CPU. At run-time it determines the architecture level, placing the value in a register. For a program wishing to validate its run-time architecture, it should be called very early in the program, if not the first macro used after control is passed to the program by the IPL process.

The APROB macro is designed to assemble and run in any standard mainframe compatible architecture system. Certain systems within the System/360 are not standard and APROB will not function on those systems. The result of the APROB macro may be compared to the preserved assembly time value supplied by the ARCHLVL macro to determine if the run time environment is supported by the program. See the section, “Architecture Levels” for the meaning of a the resulting value.

It is the responsibility of the program to decide how to handle an unexpected result unless the ANTR macro is used.

ANTR

The ANTR macro is used in conjunction with the APROB macro to pass control to specific locations based upon the architecture level established by the APROB macro. Each detectable architecture level has a macro parameter identifying the location identifies to which control passes for the detected architecture. If the architecture level detected by the APROB macro is not supported by the program, indicated by the corresponding parameter not being used, a disabled wait state results.

ANTR is specifically intended for the case where multiple architectures is supported by a program and support requires different “entry” locations for each architecture. Use of ANTR for one supported architecture is supported for validation that the run-time architecture is compatible with the program. Use of ANTR without any supporting architecture will result in a disabled wait condition.

Like APROB, ANTR is designed to assemble and run in any standard mainframe compatible architecture system. The one exception to this is the case where an invalid architecture level is presented to ANTR. In this case, the run-time architecture is actually unknown by ANTR. When entering the disabled wait state, a 64-bit extended mode PSW is used, being the most likely format compatible with the running architecture. In the case of the actual system being compatible with a System/360 system, a specification exception would result. The presentation of an invalid architecture level (one not within the range 1 to 9, inclusive) should

be considered a programming error, either in the stand-alone program or the APROB macro itself.

ARCHIND

ARCHIND sets the operator synonyms for architecture “independent” code within SATK macros depending upon the current architecture level. By default the ARCHLVL macro will internally use the ARCHIND macro to establish a set of operation synonyms that are sensitive to the assembly time architecture. To suppress this automatic behavior use the ARCHLVL parameter ARCHIND=NO. ARCHIND macro can be used independently of the ARCHLVL macro.

The set of operation synonyms defined is based upon the &ARCHLVL symbolic variable. They allow an architecture independent operation to be coded allowing actual instruction generation to be tailored for an architecture. For example in some architectures the BRANCH ON CONDITION (BC) instruction is generated and for others the BRANCH RELATIVE ON CONDITION (BRC) instruction is used. The convention is that the “generic” instruction is preceded by a '$'. So by coding $BC in the operation field, depending upon the detected architecture level, one or the other of these instructions will be generated by the single operation code. The creation of the synonyms does not effect the original instruction mnemonics and they are available for explicit use if desired. The System/370 level of instructions are used if the &ARCHLVL is zero. Unlike the &ARCHLVL symbolic variable, operation synonyms are available outside of macros.

ARCHLVL

The ARCHLVL macro is critical to nearly all SATK code. It controls how the macros generate code. It must be the first macro executed by an SATK using program. When setting the XMODE PSW setting in a program, use the ARCHLVL macro to set the XMODE setting rather than doing them separately. This ensures the level always matches the current setting.

Assembly Time

At assembly time, recognition of the ASMA target architecture is provided by the ARCHLVL macro. The macro establishes the assembly time global arithmetic symbolic variable &ARCHLVL, universally used to drive architecture specific code generation. By default, the determination is based upon the current XMODE PSW setting as specified by the initially selected Machine Specification Language (MSL) CPU definition or later establish by explicit use of the XMODE directive itself in the program. As recommended above, when explicitly setting the XMODE PSW format, let ARCHLVL do it by using the PSW parameter. This ensures the XMODE setting, the architecture level and related operator synonyms are kept consistent, thereby avoiding surprises.

The current XMODE PSW setting is exposed to a macro by means of the &SYSPSW global symbolic variable. A macro dedicated to examining this setting, ARCHLVL, will set a global

symbolic variable of the same name, &ARCHLVL, to an arithmetic value between 1 and 9, inclusive. See the section, “Architecture Levels” for the meaning of an assigned value. A value of zero indicates the XMODE setting has been disabled or is unrecognized. Only the PSWS format is not recognized by the ARCHLVL macro. Once the &ARCHLVL symbolic variable has been set, other macros utilize it to make code generation decisions.

ESA/390 Considerations

It is not possible to differentiate by PSW format between a native ESA/390 system or a z/Architecture system running in ESA/390 mode. By default, the ARCHLVL macro assumes a ESA/390 system is running on z/Architecture. To alter this assumption to being a native ESA/390 system, specify the ARCHLVL parameter ZARCH=NO.

Overriding the XMODE PSW Setting

To force a specific setting of the architecture level use the ARCHLVL parameter SET=n. 'n' in this case is a decimal integer between 1 and 9, inclusive. This parameter forces the &ARCHLVL symbolic variable to the specified value regardless of the current XMODE PSW setting. Operation synonyms, if not suppressed, will be created based upon the specified value.

The use of the SET parameter is independent from that of the PSW parameter. The program can both establish a new XMODE PSW setting and ignore it by use of the SET parameter. A good reason to do this is not apparent, but is possible.

Run Time

The assembly time architecture level can be preserved for run time by providing a label when the ARCHLVL macro is called. The symbol is used in an EQU directive setting the symbol's value to the value of the &ARCHLVL symbolic variable established by the ARCHLVL macro. In turn the symbol may be used in an address constant providing the assembly time architecture level to the program at run time.

POINTER

The POINTER macro, using the information established by the ARCHIND macro, assembles an address constant of the appropriate size for the current active architecture.

Defining Structures

Numerous structures are utilized by mainframe systems. Various macros support generation of these structures:

- ASAREA – Assigned storage area in addresses X'0' – X'1FF'.
- ASAZAREA – Assigned storage area specific to z/Architecture systems.

- DSECTS – Unifies structure generation in a single macro.
- IOCBDS – Defines the raw I/O control block
- IOFMT – Defines I/O related structures
- PSWFMT – Defines PSW format based upon the current architecture level

All macros ensure that a specific DSECT is created only once in the assembly. The DSECTS macro is recommended, but the individual macros may also be used.

Definition	Creation	Init/Term	Process
ASAREA ASAZAREA DSECTS See desc. IOFMT See desc. PSWFMT: PSWB PSWE PSWZ	ASAREA ASAZAREA	--	--

ASAREA

Defines the storage area in addresses X'0'-X'1FF' for all architecture uses. It is designed to actually generate content for the area in a CSECT or define a DSECT. In either case, a USING directive with register 0 is required. When created by the DSECTS macro, the ASA DSECT is created.

ASAZAREA

This macro is only needed to reference assigned storage areas starting at address X'11C0' on z/Architecture systems. It may generate actual content in a CSECT or define the area in a DSECT. The defined area only contains save areas. Unless a program expects to reference or use these areas, the ASAZAREA macro is not normally required even on z/Architecture systems. When created by the DSECTS macro, the ASAZ DSECT is created.

DSECTS

The DSECTS macro is a wrapper for use of all of the other structure related macros. Its primary advantage is that it uses PUSH and POP directives in managing the PRINT directive settings. It uses a single parameter, NAME, that may use a sublist to specify the structure(s) being assembled. The “Level” column indicates whether the DSECT content is sensitive to the current architecture level (see “ARCHLVL” section). The following names are supported:

NAME	Level	Description
ASA	no	Defines assigned storage areas
ASAZ	no	Defines z/Architecture save areas
CCW	no	Defines both CCW formats
CCW0	no	Defines the Format-0 Channel Command Word
CCW1	no	Defines the Format-1 Channel Command Word
CHAN	no	Defines structures used by channel-based I/O: CCW0 and CSW
CS	no	Defines structures used by channel subsystem-based I/O: CCW0, CCW1, IRB, ORB, SCHIB and SCSW
CSW	no	Defines the Channel Status Word (CSW)
FRAME	yes	Defines the default function stack frame
IO	no	Defines all I/O related structures
IOCB	no	Defines the raw I/O control block
IRB	no	Defines the Interruption-Response Block
ORB	no	Defines the Operation-Request Block
PSW	yes	Defines the PSW structure(s) in use
SCHIB	no	Defines the Subchannel Information Block
SCSW	no	Defines the Subchannel Status Word
TABLE	no	Defines the SATK table definition structures (DSECT TBL and TBLG)

The NAME=TABLE parameter requires the table.mac file.

If the NAME parameter is omitted, based upon the current architecture level the DSECTs required for the current architecture level PSW, and related I/O structures are generated in addition to the IOCB DSECT.

IOCBDS

Defines the raw I/O control block. See the “Performing Input/Output Operations” section for information on its usage. The DSECTS=IOCB parameter also defines this structure.

IOFMT

Defines various hardware related I/O structures. Uses the same name for its DSECT parameter as does the DSECTS macro for the corresponding structure: CCW, CCW0, CCW1, CSW, IRB, ORB, SCHIB, SCSW. If ALL is used, all of the I/O related structures are defined.

PSWFMT

Based upon the architecture level one or more DSECTs is created defining the PSW format:

- Level 1 – `PSWB`
- Level 2-4 – `PSWB` and `PSWE`
- Levels 5-7 – `PSWE`
- Levels 8 and 9 – `PSWE` and `PSWZ`

Changing Architecture Mode and Processor Signaling

When preparing a program for z/Architecture mode, the system, will be in the ESA/390 mode following the Initial Program Load (IPL) function. The program itself must cause the system to enter z/Architecture mode. The change in architecture mode is a special case of processor signaling. Three macros are provided for this purpose:

- `ESA390` – enters ESA/390 mode from z/Architecture mode at run-time
- `SIGCPU` – generic signaling of the same or different processor.
- `ZARCH` – enters z/Architecture mode from ESA/390 mode at run-time.

All three macros have the same register requirements:

- an even/odd pair and
- a separate register for the address of the CPU being signaled.

All three macros allow detection of the success or failure of the operation. Each of the macros offer a `SUCCESS` and `FAIL` parameter. It is recommended either or both be used to ensure detection of the success or failure of the operation. If neither parameter is used, condition code 0 indicates success. Any other condition code setting indicates failure.

Definition	Creation	Init/Term	Process
--	--	--	M: `ESA390` M: `SIGCPU` M: `ZARCH`

Side-Effects of Architecture-Mode Changes

Changing from 32-bit registers to 64-bit registers can have interesting side effects. When encountered, a program interruption usually occurs,

The first recommendation made for managing the side effects when architecture mode is changed encourages use of disabled-wait PSW's for assigned storage area new PSW's.

They can be loaded during the IPL function, particularly when using list-directed IPL by using the ASAL0AD macro. Alternatively, the program can, after controlled is passed during the IPL function, introduce these PSW's into the assigned storage area by explicitly moving them from PSW's generated in the program by the TRAPS macro. Such trap PSW's do nothing for actually correcting the problem, but do control any adverse side effects, cleanly ceasing program execution at the point of failure.

The primary issue involves signed binary integers. Systems with 64-bit general registers do not have a separate set of 32-bit general registers. Rather, the low-order 32-bits of the 64-bit general register are used for the 32-bit register when in 32-bit mode. During 32-bit mode, the sign exists in bit 0 of the 32-bit register. However, after a change to z/Architecture the sign bit exists in bit 0 of the 64-bit register. Bit 0 of the 32-bit register is now bit 32 of the 64-bit register. The CPU reset that occurs during the IPL function will set all 64 bits of the 64 bit registers to 0, but the CPU will be in 32-bit mode. The effect of this is to cause the 64-bit register's contents to be interpreted as a positive integer rather than the negative integer intended during 32-bit register mode operation.

Only the program developer knows that the bits in a register are or are not to be interpreted as a signed integer. For any signed register value whose sign needs preservation after the architecture mode change, use the LOAD (64<32) register instruction:

```
LGFR  N,N
```

This instruction will propagate the 32-bit sign into bits 0-31 preserving the register's sign. When used for this purpose the first and second operands must identify the same register, unless changing the actual register that contains the value is intended.

Side-Effects of Address-Mode Changes

Similar effects can occur when changing addressing modes. In 24- and 31-bit addressing modes, the high order bits, the first eight and first, respectively, are ignored during address calculation. The most likely situation where this can be a problem involves a base register established from use of either the BALR or BASR instructions. These instructions, and some others, can set the high-order bits of the register. As long as the register contents are used in the same address mode as was in use when the value was set, no problems occur.

When changing to a higher addressing mode, the previously ignored high-order bits become part of the address calculation resulting in the wrong addresses being used by an instruction relying upon the value established in the previous address mode. In addition to instructions causing an address mode change, the introduction of a PSW, either by the program using a LOAD PSW or LOAD PSW EXTENDED instruction or CPU during interruption recognition, can also change the address mode. The default address mode for SATK created PSW's is 24.

Again, only the program developer knows when a register contains an address and what bits should be ignored, if any. Before a change to a higher address mode, the program should ensure that the necessary high-order bits are set to 0.

Four macros are provided for the purpose of setting register bits to 0 depending upon the

current architecture level and whether the register's high-order or low-order bits or only the lower half of the register are being set to zero in the case of 64-bit registers. The following table summarizes the usage and effected bits. Each macro has the same two required parameters:

- the register whose bits should be set to 0 and
- the number of bits including the starting bit.

Macro	Level	Register Bits	Starting Bit	ESA/390 or lower	Z/Architecture
ZEROH	0-8	0-31	High starting at 0	allowed	allowed
	9	0-63	High starting at 0	prohibited	allowed
ZEROL	0-8	0-31	Low starting at 31	allowed	allowed
	9	0-63	Low starting at 63	prohibited	allowed
ZEROLH	0-8	0-31	High starting at 0	allowed	allowed
	9	32-63	High starting at 32	allowed	allowed
ZEROLL	0-8	0-31	Low starting at 31	allowed	allowed
	9	32-63	Low starting at 63	allowed	allowed

On architecture levels other than 9, ZEROLH and ZEROH are equivalent and ZEROLL and ZEROL are equivalent and may be used interchangeably.

Definition	Creation	Init/Term	Process
--	--	--	M: ZEROH M: ZEROL M: ZEROLH M: ZEROLL

When executing, ZEROH and ZEROL use z/Architecture specific instructions, targeting the entire 64-bit register. If the program has not yet changed to z/Architecture, executing these instructions will cause a program interruption. In this case use ZEROLH or ZEROLL in the source program. This situation occurs if a program is assembled with an ASMA target of -t s390x, but the program has not yet actually changed to the z/Architecture mode when running. Matching the XMODE PSW, ARCHLVL and ARCHIND settings to the running architecture ensures correct instruction generation.

ESA390

Uses SIGCPU to change architecture mode to ESA/390. Rarely needed.

SIGCPU

Signal this or another CPU. Requires the address of the CPU being signaled in a half word supplied by the `CPUADDR` parameter. If omitted, `SIGCPU` assumes the CPU is signaling itself and allocates a half word into which it stores the current CPU's address. The signaling order must be a self defining term supplied by the `ORDER` parameter.

ZARCH

Uses `SIGCPU` to change architecture mode to z/Architecture.

ZEROH

Set the high-order bits of a 32-bit (architecture levels 0-8) or 64-bit (architecture level 9) register to 0.

ZEROL

Set the low-order bits of a 32-bit (architecture levels 0-8) or 64-bit (architecture level 9) register to 0.

ZEROLH

Set the high-order bits of a 32-bit (architecture levels 0-8) or the bits starting at 32 in a 64-bit (architecture level 9) register to 0.

ZEROLL

Set the low-order bits of a 32-bit (architecture levels 0-8) or the bits starting at 63 and ending with bit 32 in a 64-bit (architecture level 9) register to 0.

Terminating the Program

The only way to terminate a bare-metal mainframe program is to enter a “disabled-wait” state. A disabled wait state results when a PSW is introduced as the active PSW with the wait bit set, causing instruction execution to cease while waiting for an interruption, combined with disabling of interruptions by the same PSW. Because the masked interruptions are disabled, they will not be recognized if they occur and instruction execution will not continue. The combination of these flags in the PSW effectively halts the processor.

Two convenience macros are provided for the assembling of such PSW's:

- `DWAITEND` – for normal termination and
- `DWAIT` – for abnormal termination.

Both macros will generate a PSW properly formatted 64-bit PSW for creating a disabled wait

state and will optionally introduce in-line the PSW as the active PSW by issuing the `LPSW` instruction regardless of the architecture. LOAD PSW functions properly for this purpose on all architectures. If the program loads a PSW created by `DWAIT` or `DWAITEND` it must also use a `LPSW` instruction. The program must not use `LPSWE` or `$LPSW`.

By convention, the instruction address of the disabled-wait state PSW will provide rudimentary information about the cause of the termination.

Definition	Creation	Init/Term	Process
--	--	M: `DWAIT` M: `DWAITEND`	--

DWAIT

This is the foundation for creating disabled wait state PSW's and optionally loading it. The macro uses the conventions for identifying SATK vs. program specific conditions described in the “Disabled Wait States” section previously. `PGM` and `CODE` parameters are hexadecimal digits, generating an address of PPCCCC, where 'P' is supplied by the `PGM` parameter and 'C' is supplied by the `CODE` parameter.

DWAITEND

This macro is equivalent to:

```
DWAIT PGM=0,CODE=000
```

Flying Without a Net

Formal ISR's, while necessary in some settings, can be overkill in others. ISR's are critical when multiple concurrent activities are in use. ISR's capture asynchronous events for later activity handling and recognize when an activity has a program issue. For a bare-metal program performing a single activity, the program itself can capture these events without the need of formal ISR's. This does not mean that interruptions do not happen, but they are handled directly by the program, that is, without formal ISR's.

There exists no mechanism to disable any class of interruptions from occurring when the right conditions exist for the interruption being accepted by the CPU. This is particularly true when things go awry. In some cases, the program needs to capture the event. That being said, each interruption class needs to have a valid new PSW established for it and in those cases where the program needs to recognize the event, the new PSW needs to support it. The following macros are used to facilitate the capturing of expected and unexpected interruptions

- `TRAPS` – Run-time enabling at assigned storage locations of disabled wait state new PSW's for unexpected interruptions.

- TRAP64 – Creates disabled wait state PSW's inhibiting further action for unexpected interruptions for the class of architectures that utilize 64-bit PSW's (all pre-z/Architecture systems).
- TRAP128 – Creates disabled wait state PSW's inhibiting further action for unexpected interruptions for the class of architectures that utilize 128-bit PSW's (currently only z/Architecture).
- ASALOAD – Establishes disabled wait state new PSWs before entering the program during the Initial Program Load function. Useful in IPL contexts not utilizing an IPL device. Device specific installation considerations required when preparing IPL media.
- ASAIPL – Establish an IPL PSW within the assigned storage area for program entry during the manually initiated Initial Program Load function.
- CPUWAIT – Wait for an expected event using a new PSW that passes control where and in which selected state required by the program.

Definition	Creation	Init/Term	Process
--	ASALOAD ASAIPL TRAP64 TRAP128	Init: TRAPS	M: CPUWAIT

TRAPS

Early in the execution of the program, valid PSW's for each interruption class can be established by using the TRAPS macro. TRAPS creates disabled-wait state PSW's and moves them to assigned storage locations allowing any interruption to be captured and ensuring an endless PSW loop does not result. Its actions are based upon the assembly current architecture level as reported by the ARCHLVL macro. Uses TRAP64 and TRAP128 to generate disabled wait PSW's as needed.

Levels 1-9 TRAP64	Level 9 TRAP128	Location Description
X'0'-X'7'	X'1A0'-X'1AF'	Restart New PSW
X'58'-X'5F'	X'1B0'-X'1BF'	External New PSW
X'60'-X'67'	X'1C0'-X'1CF'	Supervisor-Call New PSW
X'68'-X'6F'	X'1D0'-X'1DF'	Program New PSW
X'70'-X'77'	X'1E0'-X'1EF'	Machine-Check New PSW
X'78'-X'7F'	X'1F0'-X'1FF'	Input/Output New PSW

The ENABLE parameter controls the actions of the TRAPS macro:

- ENABLE=YES causes the trap PSW's to be moved to their assigned storage locations from those generated by the TRAPS macro itself.
- ENABLE=NO causes the TRAPS macro to generate in-line the disabled wait PSW's intended for use by the TRAPS ENABLE=ONLY option.
- ENABLE=ONLY cause the PSW's created by the separate TRAPS ENABLE=NO option to be moved to their assigned storage locations.

TRAP64

Generates in-line 64-bit disabled wait state PSW's in the current architecture format or as specified by the PSW parameter. Each disabled-wait PSW has as its instruction address the assigned storage location of its corresponding Old PSW.

Because the Restart New PSW assigned storage location is not contiguous with the other New PSW's, it needs special options depending upon the context in which the PSW's are being generated.

- RESTART=YES causes the Restart New PSW to be included with the generated PSWs.
- RESTART=ONLY causes only the Restart New PSW to be generated, inhibiting the other New PSW's.
- RESTART=NO causes the other PSW's to be generated without the Restart New PSW.

TRAP128

Generates in-line 128-bit disabled wait state PSW's in z/Architecture format. Each disabled-wait PSW has as its instruction address the assigned storage location of its corresponding z/Architecture Old PSW.

ASALOAD

The ASALOAD macro initiates a new region with a starting address of 0, causing its contents to be placed at absolute 0. It allows, during the IPL function, initialization of the new PSW assigned storage locations trapping interruptions for which the program is not prepared to support. The value of doing this during the IPL function is that early program errors, including an incorrect IPL PSW can be caught.

The region created by the ASALOAD macro is always 512 bytes in length.

Unless the entire physical image is loaded into storage, the placement of the region within the image does change the initialization of the assigned storage area. If the entire image is being loaded into storage, the image must be loaded at absolute address 0 and the region created by the ASALOAD macro must be the first region of the image.

Refer to the *ASMA* manual for details on the relationships of regions, control sections and the image.

ASAIPL

The `ASAIPL` macro is intended to inject into the ASALOAD created region a valid PSW at absolute address 0. This PSW is suitable for the use with either the IPL or restart functions.

`ASAIPL` must be preceded in the assembly by an ASALOAD macro creating the region and control section containing the assigned storage location content.

The PSW placed at the start of the `ASALOAD` created region overwrites the restart trap PSW created by the `ASALOAD` macro itself at absolute address 0. If the program wishes to trap any subsequent attempts to restart the program with a restart interruption, subsequent to control being passed to the program, it must overwrite the IPL or restart new PSW with a trap PSW. The `TRAP64` macro can create the PSW by use of the `RESTART=ONLY` parameter. The program can then move this PSW to absolute addresses 0-7, inclusive.

CPUWAIT

On occasion a program without the complexity of interrupt service routines may need to wait for specific events expected by the program. These events are:

- expired timers, and
- input/output operations.

If the program does not have any interrupt service routines, how can this be accomplished? At the level of the interruption an interrupt service routine is nothing more than a place to which control is passed to handle the interrupt. Because such events can occur asynchronously from the running program, interrupt service routines typically require saving upon entry and a restoration of the program's state. By causing the interruption to pass control to a place where the program's state is known, only the passing of control is required.

If a program expects to wait, the place at which its state is known is at the point in the program where it expects to perform the wait. In other words, inline, is the best place to perform the wait. The CPUWAIT macro performs an inline wait. It does this by:

1. establishing a new PSW for the interrupt class upon which it is waiting that passes control following the instructions, and optionally saving the previous new PSW;
2. loading an enabled wait PSW for the anticipated event; and
3. receiving program control following the event
4. passing control as specified by the class' related parameter,optionally restoring the previous new PSW before doing so.

The need to save program state is eliminated because the program ceases instruction execution. The formal interrupt service routine is eliminated because the instructions

following the CPUWAIT macro process the event.

Two events are supported by the CPUWAIT macro:

- External interruptions and/or
- Input/Output interruptions.

Both are supported by the same CPUWAIT macro depending upon its parameters. This allows for example, with the appropriate preparation by the program to wait for an input/output event or a timeout if the input/output event doesn't occur.

In either case the program continues execution following the CPUWAIT macro. The CPUWAIT macro passes control as specified by its parameters. It also allows the program to differentiate between the interruption classes by setting the condition code:

- 1 – an external interruption occurred, or
- 2 – an input/output interruption occurred.

The parameters and their relationships are summarized below.

External	Input/Output	Description
EXT	IO	Describes whether and how control is passed for the interruption • YES – control is passed to the statements following the CPUWAIT macro • NEW – uses the EPSW or IPSW as a template for the required state upon returning to the program and explicitly uses the PSW for passing control by issuing a LOAD PSW or LOAD PSW EXTENDED instruction. • label – branches to the label if the interruption occurred If omitted the interruption class is not enabled and any interruptions remain pending.
EPSW	IPSW	Provides a PSW template for the interruption class new PSW. The template PSW except for its: • interruption masks, and • condition code. Required if EXT=NEW or IO=NEW specified.
ESA	ISA	YES causes the interruption class' new PSW to be saved and restored before and after waiting, respectively. NO does not preserve the previous new PSW. Defaults to YES.

External	Input/Output	Description
AM		If EXT or IO specifies either YES or label, this address mode is used. Defaults to 24-bit addressing. If EXT or IO specifies NEW, the template PSW's address mode is used.
RELO		Specify YES to cause relocation if EXT=YES or label, or IO=YES or label. Ignored for EXT=NEW or IO=NEW. Defaults to NO. Relocation of the template PSW instruction address when either EXT=NEW or IO=NEW is the responsibility of the program before it is used by the CPUWAIT macro.
REG		A register required for use by the CPUWAIT macro. Defaults to 1.
Not applicable	CHAN	Allows the waiting PSW's channel masks to be specified for System/360 or System/370 BC-mode environments. Defaults to 254 (all channels enabled).

The program should examine any interruption information supplied in assigned storage locations related to the interruption class. Presumably the CPU is placed into a wait state for the purpose of waiting for an event to occur. Generally, because the interruption classes supported by the CPUWAIT macro may occur for multiple reasons, only by examination of the interruption information is it possible to determine if the expected event occurred.

Performing Input/Output Operations

Mainframe systems from their inception are designed to perform input/output operations in an interruption driven environment. Interruptions result from initiated input/output operations and other input/output related notifications occur through interruptions. While it is technically possible to perform input/output operations without utilizing interruptions, it is not possible to be informed of input/output related notifications without interruptions. The use of the input/output system without interruptions has never been encouraged. SATK therefore only supports the input/output system utilizing interruptions.

Interruptions resulting from initiated input/output operations are referred to as “solicited” interruptions. Event notification interruptions outside of input/output operations are referred to as “unsolicited” interruptions.

Mainframes have utilized two different architectures for the input/output system. Both are based upon the concepts of channel programs executed by the channel with the aid of the devices themselves. From the view of the CPU, Initiating and termination of the channel programs differs between the two architectures. The form and structure of input/output notifications also differ. Providing a simple, consistent interface to the two different I/O architectures without losing flexibility is the primary goal of the supplied macros.

The first architecture, used individual channels controlled by the program for input/output operations. By the third generation of mainframes, this system was replaced by a channel subsystem that managed all of the physical channels and performed input/output operations on behalf of the program. As the technology advanced, the information necessary for the channel subsystem to perform its tasks increased. New storage based structures were introduced for this purpose.

The macros have no knowledge of the operations, channel commands or details of successful or failed activities. The program (and developer) must supply such knowledge. Under either architecture, the macros will **initiate** an operation and **detect** its success or failure as defined by the program under either architecture. In very simple settings, the input/output operations can be transferred between the two input/output architectures without program changes. See the `hello.asm` sample program for an example of this usage. Because facilities differ between the two architectures, this transportability is not generally likely for complex operations.

The input/output operations typically involve three aspects of the input/output architecture:

- preparing the CPU for input/output operations,
- making individual devices ready for use and then
- using the device for operations.

Seven macros work together for these functions:

- `IOINIT` – prepares the CPU for input/output operations, typically used once by the program;
- `IOCB` – the assembled structure used to define an input/output device used by the program, used once for each device;
- `IOCBDS` – creates a DSECT named `IOCB`, defining the `IOCB` structure fields, used once by the program;
- `ENADEV` – makes devices defined by an `IOCB` macro ready for use by the CPU;
- `RAWIO` – performs an inline input/output operation with an `IOCB` defined device; and
- `IOTRFR` – calculate the number of bytes actually transferred following a successful I/O operation.
- `RAWAIT` – performs an inline wait for an unsolicited interruption from the IOCB defined device.

All, except `IOINIT` utilize a common structure in performing these function.

The generated `IOCB` structure links the program, the input/output system and actions with the device. Because the `ENADEV`, `RAWAIT`, and `RAWIO` macros are strictly driven through the IOCB a single use of the these two macros can be used for all devices and all input/output operations by simply addressing a different IOCB before executing the inline instructions.

This is intended to allow these three macro to be placed in a subroutine and reused by the program.

Definition	Creation	Init/Term	Process
IOCBDS	IOCB	Init: IOINIT Init: ENADEV	M: IOTRFR M: RAWIO M: RAWAIT

All of the macros, with the exception of IOCBDS, are dependent upon the current architecture level implied by the current XMODE PSW setting. The architecture level is used to determine which I/O architecture scheme is in use. Channel command words can utilize either of two formats. Only the IOTRFR macro is sensitive to the CCW format specified by the XMODE CCW setting.

IOCB Structure

The IOCB structure links all of the macros (with the exception of the IOINIT macro) to the program and each other. The IOCB is the mechanism by which a SATK program and the input/output related macros communicate with each other and attached devices. The IOCB also normalizes the differences between the two input/output architectures. The IOCBDS macro identifies certain fields within the structure as read-only. The program should not alter these fields. They are critical to the operation of the macros.

The two input/output architectures both depend upon assigned storage areas and the use of channel-command words for communication with a device performing an input/output operations. Each uses a similar, although not identical, structure for communicating the results of an operation. Locating this information differs, but its location is made transparent to the SATK program through use of the IOCB structure.

The channel subsystem specific structures are not inherently part of the IOCB structure, although the IOCB macro can as a convenience create the Operation-Request Block (ORB) and reserve space for the transient needs of the program for access Subchannel-Information Block (SCHIB) and Interrupt-Response Block (IRB) access. Even when the IOCB provides these structures, they are external to the IOCB structure. The addresses for each are used by the I/O macros as if the structures were created separate from the IOCB. Whether a field is read-only or not, the information within the IOCB structure is shared between the I/O macros and the program.

The IOCB structure is assembled by the IOCB macro and defined by the IOCBDS macro. All fields may be examined by the program, but only specific fields should be modified by the program. These fields are identified by an 'X' in the IOCB dummy section created by IOCBDS. Intentionally the IOCB structure may be completely specified during assembly or completely created during program execution or some combination. Regardless of the creation, each device using the I/O macros described in this section requires an IOCB structure.

The IOCB is utilized for multiple purposes related to the input/output system. Some fields are expected to be modified by the program as needed. These

- Provide the logical identification of the corresponding device or subchannel as known by the program, field `IOCBDEV`;
- Identify the input/output operations to be performed, fields `IOCBCAW` or `IOCBORB`;
- Specify conditions related to the device or subchannel operation representing errors, of special significance or completely ignored, fields `IOCBUM`, `IOCBCM`, and `IOCBWAIT`; and
- Locate structures required by the channel subsystem used by the `ENADEV`, `RAWIO` or `RAWAIT` macros, fields `IOCBIRB` and `IOCBSIB`.

Other fields, are expected to only be examined by the program, but not modified. These fields:

- Provide the corresponding I/O system designation for the device or subchannel when interacting with it, initialized by `ENADEV`, field `IOCBID`;
- Normalize I/O architecture specific interruption information into the IOCB structure for examination by the `IOTRFR`, `RAWIO` or `RAWAIT` macros, fields `IOCBSCCW`, `IOCBSCNT`, `IOCBUS`, and, `IOCBCS`; and
- Internally used by `RAWIO` or `RAWAIT`, fields `IOCBZERO`, `IOCBUT`, `IOCBCT`, `IOCBSC`.

The following table summarizes the relationships between the various macros, the IOCB fields and the program. The column heading 'CH' indicates the field is used by channel-based input/output system. The column heading 'CS' indicates that the field is used by the channel subsystem.

IOCB Field	Program	CH	CS	IOCB Parm.	ENADEV	RAWIO	IOTRFR	RAWAIT
IOCBDID	RO	X	X		initializes	input		input
IOCBDEV	RW	X	X	DEV	input			
IOCBUM	RW	X	X	UERR		input		
IOCBCM	RW	X	X	CERR		input		
IOCBUS	RO	X	X			updates		updates
IOCBCS	RO	X	X			updates		updates
IOCBUT	RO	X	X			updates		updates
IOCBCT	RO	X	X			updates		
IOCBSC	RO		X					updates
IOCBWAIT	RW	X	X	WAIT				input
IOCBSCCW	RO	X	X			updates	input	

IOCB Field	Program	CH	CS	IOCB Parm.	ENADEV	RAWIO	IOTRFR	RAWAIT
IOCBSCNT	RO	X	X			updates	input	
IOCBCAW	RW	X	***	CCW, KEY, FLAG *		input		
IOCBORB	RW	***	X	ORB		input		
IOCBIRB	RW		X	IRB		input **		input **
IOCBSIB	RW		X	SCHIB	input **			

* IOCB macro parameters CCW, KEY and FLAG are used to initialize the generated ORB when an explicit ORB parameter is omitted.

** The field is not altered by the macro. However, the area to which it points is updated by the macro.

*** The IOCBCAW and IOCBORB fields overlap or correspond. Both fields serve to identify to the I/O system the location of the channel program constituting the initiated input/output operations depending upon the system in use. The IOCBCAW directly locates the channel program. The IOCBORB field indirectly locates the channel program by providing the location of the ORB that contains the explicit location.

Using the Input/Output Macros

This section describes the steps required in a program to use the basic input/output macros. The following two sections give details related to the use of the macros in a specific input/output architecture.

The RAWIO and RAWAIT macros depend upon detecting a solicited or unsolicited interruption, respectively, from the targeted device. Interruptions from any other device are ignored and lost. See the input/output architecture considerations concerning the IOINIT macro for details on managing interruptions.

Step 1 – Define the Input/Output Structures to the Assembly

Assemble the input/output related dummy sections:

- DSECT macro omitting the NAME parameter, selecting architecture sensitive dummy sections,
- DSECT NAME=(CH,IOCB), for a channel I/O system,
- DSECT NAME=(CS,IOCB), for a I/O channel subsystem, or
- DSECT NAME=IOCB, equivalent to using the IOCBDS macro itself.

Step 2 – Assemble Device Definitions

Use the `IOCB` macro to define the devices the program plans to use. IOCB definitions are static. The macros described here do not perform device discovery involving the use of the `SENSE ID` channel command. There is nothing stopping the program from using a dynamically altered IOCB for that purpose.

Device definitions may be placed anywhere in the assembly. Each macro that uses the IOCB structure expects to address the contents via a `USING` directive for the `IOCB` dummy section.

If the channel program consists of the same input/output operations it may be assembled with the device definition.

Step 3 – Prepare the CPU for Input/Output Operations

Use the `IOINIT` macro to prepare the CPU to accept interruptions from various sources. The `IOINIT` macros loads the appropriate control register for this purpose. By default, all input/output channels or subchannels are enabled for interruption recognition by the CPU.

Step 4 – Enable Each Device for Input/Output Operations

For each device, use the `ENADEV` macro to prepare the device for use. By incorporating the `ENADEV` macro in a subroutine, a single use of the macro may be used for each device by simply loading the register used to address the IOCB dummy section with each IOCB's address.

Step 5 – Perform an Input/Output Operation

The `RAWIO` macro initiates the execution of a channel program by a device or subchannel using the IOCB structure associated with the device. It performs the operations and accumulated solicited interruption information for analysis of its success or failure. Special handling of a physical end-of-file condition (`EOF` parameter) or need to access device sense data (`SENSE` parameter) are provided. If either are used the unit exception or unit check conditions, respectively, must not be set in the IOCB `IOCBUM` field or `IOCB` macro `UERR` parameter.

Depending upon the needs of the program the IOCB fields may be updated for the specific operation. Different operations with a device or subchannel may require different handling of the status indicators. Some may be errors in some settings but not others. Program alteration of the `IOCBUM` or `IOCBCM` fields may be required. Correspondingly, a different sequence of input/output operations may require changes to the IOCB or channel subsystem structures to utilize a different channel program by alteration of the `IOCBCAW` or the address of the first channel command word in the channel subsystem Operation-Request Block.

For input/output operations where the number of bytes transferred is unpredictable, the `IOTRFR` macro calculates the number of bytes actually transferred based upon the accumulated status information in the IOCB. The macro assumes a successful completion of

the input/output operation. This usually requires suppression of incorrect length indication. Non-default IOCBUM, CCW and ORB flags may be required to properly utilize the indication.

Step 6 – Detect Device Attention or Other Event

Some devices utilize the attention unit status to indicate that input is available. This is usually the case for devices used by an operator, for example a console or graphic display unit. Use the RAWAIT macro to perform this function. The macro waits for an unsolicited interruption from the device with any of the status indications set by the IOCB macro WAIT parameter or contained in the IOCBWAIT IOCB field at the time the RAWAIT macro is executed by the program.

By default, the IOCB WAIT parameter sets the attention status.

Channel Input/Output Considerations

No special considerations exist for use of the IOCBDS, ENADEV, or IOTRFR macros.

IOCB

The IOCB macro positional macro parameter DEV specifies a device's channel and unit address.

The address assigned to channel attached device when using the Hercules emulator is the address provided in the device's configuration statement.

IOINIT

Control register 2 is loaded by the IOINIT macro on System/370 or System/380 systems.

The CPU reset that is part of the IPL function will set all bits in control register 2 to ones. If all channels are to be enabled for interruptions, then IOINIT is superfluous following an IPL.

System/360 systems do not support control registers and on these systems, IOINIT generates no instructions.

Managing I/O Interrupts on System/370 in EC mode or System/380

To ensure interruptions are not lost though, a channel must not be enabled for interruptions if the program is not prepared to lose the interruption. At the level of an interruption, the CPU is unable to differentiate between solicited or unsolicited interruptions. If the CPU is enabled for input/output interruptions by the PSW, and an interruption occurs from a device that is on a channel that is enabled for interruptions, the CPU will accept it. If the channel is not enabled for interruptions, the interruption will remain pending until such time as it is enabled.

By separating a device or set of devices onto its own channel, the program can control which devices have the potential for lose. Devices used exclusively with RAWIO, can generally be safely used without concern. Unsolicited interruptions though can occur at any time in

relationship to other operations. Hence devices which may require use of the RAWAIT macro should be separated. In this setting IOINIT can manage which channels can present an interrupt and which can not. Before issuing the RAWAIT macro, IOINIT can enable the channel on which the device resides and disable the others. Before issuing the RAWIO macro, IOINIT can be used to disable interruptions from these devices and enable them for the other devices used for normal operations. In the case of a solicited operation following the detection of an unsolicited event, no change in enabled channels is required.

More advanced management of interruptions requires formal interrupt service routines.

Managing I/O Interrupts on System/370 in BC Mode

The same considerations apply to System/370 in BC mode. However, the use of channel masks in control register 2 only apply to channels 6 or above. For channels 0 through 5 inclusive the techniques described in the following section apply.

Managing I/O Interrupts on System/360

The same considerations apply to System/360 as on System/370. However, the “knob” that controls channel interrupts is within the PSW itself. The PSW enabling input/output interruptions is embedded within the CPUWAIT macro used by the RAWIO or RAWAIT macro. Run-time modification of the mask within this waiting PSW is not readily available. In this case separate RAWIO or RAWAIT macros should be used with the appropriate enabling channel mask set by the CHAN parameter available on the two macros.

RAWIO

Input/output operations are initiated at the device by means of a START IO instruction. The CPU then waits for solicited interruptions until the accumulated status provided by the CSW indicates the channel end and device end status.

Any interruption that occurs for the target device is considered a solicited interruption by the RAWIO macro. No mechanism exists to differentiate between solicited and unsolicited interruptions for channel attached devices. Only the state of the program knows which is expected.

RAWAIT

Any interruption that occurs for the target device is considered an unsolicited interruption by the RAWAIT macro. No mechanism exists to differentiate between solicited and unsolicited interruptions for channel attached devices. Only the state of the program knows which is expected.

Multiple status conditions are possible for unsolicited interruptions besides the attention unit status.

Channel Subsystem Considerations

No special considerations exist for use of the `IOCBDS` or `IOTRFR` macros. While simple scenarios may not require an understanding of the roles of the channel subsystem specific structures, generally programmer familiarity will be required.

IOCB

The `IOCB` macro positional macro parameter `DEV` specifies a subchannel's device number. The device number is analogous to a channel attached device's channel and unit addresses.

The subchannel device number when using the Hercules emulator is the address provided in the device's configuration statement.

The areas identified by the `SCHIB` or `IRB` parameters must be large enough to contain the structure that may be stored. For the SCHIB, this is 13 full words, 52 bytes, full word aligned. For the IRB this must be 24 full words, 96 bytes, full word aligned. It is possible for all IOCB's to share the same 96-byte area for their respective SCHIB or IRB because at any given point in time the area will be used either for only a single SCHIB (by the `ENADEV` macro) or IRB (either the `RAWIO` or `RAWAIT` macro) related to one subchannel.

IONIT

Control register 6 is loaded by the `IOINIT` macro on systems having a channel subsystem with a mask enabling all subchannel subclasses for interruptions. Control register 6 is set to all zeros by the CPU reset that occurs during the IPL function. Without a setting of mask bits in control register 6, no subchannel interruptions will be recognized by the CPU. `IOINIT` is required when a channel subsystem is in use.

Managing Subchannel Interruptions

The potential for interruption loss exists with channel subsystem interruptions in the same way with channel attached devices. Because the channel subsystem takes over responsibility for management of the channels, interruptions are managed through the concept of an interruption subclass. When the I/O reset associated with the IPL function occurs, all subchannels are placed in interruption subclass 0. Interruption subclasses are limited to eight, subclasses 0 through 7 inclusive. By adjusting the enabled interruption subclass in control register 6, the program can control which devices are enabled at any given time. By disabling an interruption subclass, all subchannels assigned to the subclass remain interruption pending until the CPU is enabled for input/output interruptions and the interruption subclass is enabled by its mask bit being set to one in control register 6. As with channel attached devices, the interruption mask can be manipulated by the `IOINIT` macro.

See the ENADEV considerations for setting a subchannel's interruption subclass.

ENADEV

The operation of ENADEV on channel subsystem is more complex than for channel attached devices. Each subchannel in the channel subsystem is examined by using the `STORE SUBCHANNEL` instruction until the device number, IOCB `IOCBDEV` field, is found or until all subchannels have been examined. Once the subchannel with the device number is located, the `IOCBDID` IOCB field is initialized with the subchannel's subsystem identification number. The subchannel itself is enabled with the channel subsystem using a `MODIFY SUBCHANNEL` instruction.

By default the subchannel will utilize an interruption subclass of zero. The interruption subclass can be modified when the subchannel is enabled by supplying a value in the register indetified by the `ENADEV CLS` keyword parameter. If the `CLS` parameter is omitted, the interruption subclass remains unchanged. See the section “Managing Subchannel Interruptions” why this may be useful.

By specifying the `CSNS=YES` parameter, the subchannels enabled by the macro will be enabled for concurrent sense. Separate macros must be use to enable a mixture of some subchannels enabled for concurrent sense and some that are not.

The strategy used by the `ENADEV` macro in locating the subchannel associated with a device number is not encouraged for many devices. Rather than searching the subchannels for each desired device number, it is more efficient to scan the subchannels once and repetitively scan a memory resident table containing the device numbers sought by the program until each is found or the subchannels are exhausted. For a small number of devices this strategy is simpler if less efficient.

RAWIO

Input/output operations are initiated at the subchannel by means of a `START SUBCHANNEL` instruction. The CPU then waits for solicited interruptions from the subchannel until primary status is presented by the SCSW stored with the IRB by the `TEST SUBCHANNEL` instruction. Interruptions occurring subsequent to the `START SUBCHANNEL` instruction are assumed to be solicited by the `RAWIO` macro.

Depending upon the nature of the input/output operations a program supplied Operation-Request Block, ORB, may be required. The ORB macro may be used to assemble the structure.

If the program elects to use different channel programs with the subchannel with which an IOCB is associated, the ORB must be updated by the program before the `RAWIO` macro is used regardless of whether it is supplied by the program or supplied by the `RAWIO` macro.

Use of concurrent sense is possible with the `RAWIO` macro provided the subchannel has been enabled for it through use of the `ENADEV CSNS=YES` parameter. The location to which control is passed for unit check device status must be prepared to examine the IRB, located by the IOCB `IOCBIRB` field, for concurrent sense. The assumption is that the sense

information is available in the stored IRB. The SCSW bit 14 and the ERW bit 7 must both be one for concurrent sense information to be present in the IRB ECW. Detailed formats for areas other than the SCSW must be supplied by the programmer.

RAWAIT

The `RAWAIT` macro waits for an interruption from the target subchannel. Any interruption from the target subchannel for which the SCSW device active status is present is treated as an error.

Only the attention unit status condition is reflected to the program via an unsolicited interruption. Other status conditions present with channel attached devices are handled directly by the channel subsystem.

function.mac

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 for s390 CPU's. Macros for the use of such conventions in assembler programs are provided in the file `function.mac` found in the `srcasm` directory. This section describes these conventions, how they have been adapted and implemented by SATK.

The ELF file format is the standard used by the UNIX family of operating systems for representing unlinked object modules and linked executable programs. Each processor/system supporting this file format also provides an Application Binary Interface (ABI) Supplement defining how the file format is adapted to the specific processor. The ABI also defines how program languages implemented for the specific system will compile structures, manage a function call stack and perform function calls. The ABI is closely tied to the C programming language, but its use is not restricted to C. Both the ELF standard and the ABI supplement are of significant interest to compiler and operating system developers that utilize the two references.

Because SATK 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 with the GNU assembler are ABI compliant. The GNU assembler, `as`, does actually support the ELF file format.

The SATK implementation of functions relies heavily on these references:

- *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

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. The program state, regardless of the number of subroutine calls, is preserved in

the program stack. 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.

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 variable data in CPU registers to be preserved, 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 program is running as local data. The second form, CPU registers, must be transferred to storage, also in the stack frame, to preserve it, and returned to the registers 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 the 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 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`
		Init: `STKINIT`	M: `CALL`
			M: `CALLR`
			M: `RETURN`

The Stack Frame

As defined by the ELF ABI supplements, the stack frame consists of three areas, only one of which is required:

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 global to the program. All functions will expect the same 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. 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 following the `ARCHLVL` and `ARCHIND` macros.

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, 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.

SATK Function Register Conventions

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.

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

Registers	Levels	Register Usage Description	Volatile
R0	all	Available for local function usage	yes
R1	all	Available for local function usage	yes
R2-R5	all	On function entry, optional binary integer parameter	yes
	all	On function exit, optional binary integer returned value	
R6	all	On function entry, optional binary integer parameter	no
		After entry, available for local function usage	
R7-R12	all	Available for local function usage	no
R13	all	Function base register	no
R14	all	Function return address	yes
R15	all	On function entry, caller's stack frame pointer	no
	all	After function entry, function stack frame pointer	
F0, F2	all	On function entry, optional floating point parameter	yes
		On function exit, optional floating point returned value	
F4, F6	all	Local function floating point values	no
F1, F3, F5, F7-F15	7-9	Local function floating point values	no

Registers	Levels	Register Usage Description	Volatile
FPC	7-9	Available for local function usage	yes
A0-A15	6-9	Available for local function usage	yes
C0-C15	2-9	Global system control values	yes

Alterations to these conventions are possible. However, the use of the SATK supplied macros requires that R13-R15 are not modified. Use of registers R13-R15 is enforced by use of the macros contained in `function.mac`. The one register whose content must not be altered before a function is called is R15. The value of the stack pointer upon entry to a function must be the same when a function is called. It is recommended that use of R15 be avoided when using SATK functions.

The following sections provide additional details with regard to specific registers and how their SATK usage may differ from those identified in the ELF ABI references.

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 generalizes the use of R12.

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 all have the ability to use the PSW instruction address for 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

A function's return address is placed in R14. Generally, R14 should be avoided. However, if a function must use R14, it must restore its value before using its `RETURN` macro.

R15

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.

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. While the ELF ABI supplements provide space for the calling function's general and floating point registers, 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.

PACK	FP	AFP	Registers	Reserves Space
NO	NO	NO	0	
NO	NO	YES	0	
NO	YES	NO	4	F0, F2, F4, F6
NO	YES	YES	16	F0-F15
YES	NO	NO	0	
YES	NO	YES	0	
YES	YES	NO	2	F4, F6
YES	YES	YES	14	F1, F3, F4-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 and space is not reserved 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.

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 variables 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 functions 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 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 the the function being called.

If the parameter is omitted, the location of the function being called is assumed to be in general register 1.

Supporting Multiple Architectures in the Same Program

The most direct path to supporting multiple architectures, while minimizing coding differences is through the use of functions. The function macros discussed above provide basic support of functions in an assembly targeting a single architecture. Utilizing functions in a program supporting multiple architectures involves extending the definition and calling of functions in a way that allows architecture specific versions of a function to exist along with versions that may be shared between architectures. This is really about utilizing unique names. Actually reusing the coded function requires the ability for the overloading of the function name as coded in the source program to be reused in these different contexts while being differentiated from another use of the name. As with functions themselves that implement a set of subroutine call conventions, supporting multiple architectures in the same program involves a set of coding conventions that overload the name with some attributes.

To use the same coded function in multiple contexts requires the function definition, and use of local variables, to be wrapped in a macro. This wrapper macro definition must utilize a macro that understands the naming conventions for functions of certain types.

Additionally supporting multiple architectures dictates the use of targets and command-line arguments tailored for multiple architecture development. The source program itself must be structured with multiple architectures in mind. A number of moving parts are involved in creating a program that can be used in multiple architectures.

See the section “Coding for Multiple Architectures” for details. It encapsulates the compatibility requirements identified below.

Compatibility Requirements

A program that supports multiple architectures must ensure for a given instance of use that:

1. Instructions used in the program by its different architectures must be recognized by the ASMA target. Having the assembler fail to recognize an instruction causes the build to fail.
2. The IPL PSW is compatible with the run-time platform. A specification exception when introducing the IPL PSW is not a desirable result of the IPL function.
3. Structures shared by the program in shared portions of the code must be compatible with the run-time platform. Addressing exceptions can otherwise occur.
4. Code specific to an architecture is only used by that architecture. Various unexpected

results can occur otherwise.

Underlying aspects of each of the mainframe architectures and the SATK implementation drive the compatibility rules.

Assembler Instruction Recognition

Only three target architectures guarantee that all instructions are recognized by the ASMA assembler: `24`, `31` and `64`. Programs supporting multiple targets should use one of these targets to ensure assembly recognition of instructions specific to any supported architecture. SATK's use of the `ARCHIND` operation synonyms makes this mandatory.

Compatible IPL PSW

The IPL PSW must be compatible with the run-time platform. Separate IPL media regardless of the form it takes (a list-directed IPL directory or an emulated device) is required for each supported case where the IPL PSW is not compatible.

Fortunately, the 64-bit extended format PSW is supported by all architectures from System/370 (level 3 architecture) through ESA/390 on a z/Architecture system (level 9 architecture). The only outliers are the System/360 (level 1 architecture) and System/370 in basic control mode (level 2 architecture). If the outliers are not supported a single build can support all architectures used by the program from 3 through 9. z/Architecture starts out in ESA/390 mode (level 8 architecture) following its IPL.

Address mode is more selective. However, the IPL function from an emulated device is constrained to the first 16 megabytes of storage because of the use of Format-0 CCW's by the IPL function.

The 24-bit address mode constraint does not exist for the list-directed IPL case. The choices are either 24- or 31-bit addressing, the only address modes supported by a 64-bit PSW. If all of the supported architectures support 31-bit addressing, the IPL PSW may enable 31-bit addressing and the loaded program may reside within the first two gigabytes of memory.

For simplicity sake and the greatest flexibility, 24-bit address mode is recommended. This requires the program, at least at entry, to reside below the 16-megabyte boundary.

Regardless of the choice, the initial `XMODE PSW` setting can be forced by use of the ASMA command-line argument `--psw` independent of the selected assembly target. By using this command line argument, the source program can use the `PSW` directive to correctly build the IPL PSW without coding changes. The PSW format of the lowest architecture with a compatible IPL PSW should be selected by the `--psw` argument.

Sharing Structures

The major influence on structure sharing involves fields intended for different register sizes. While use of fields smaller than a register requires appropriate instruction selection, these can frequently be accommodated by use of operation synonyms supplied by the `ARCHIND` macro.

However, fields that accommodate full register data may or may not be compatible depending upon how the field is constructed. Fields expected to be used with 64-bit registers are eight bytes in length. Fields expected to be used with 32-bit registers are four. If the fields are designed to match the register size, then the same structure targeted for 32- or 64-bit registers will have different displacements for a field with the same logical usage. This results in two different dummy sections with differently named fields for the same logical structure. A logical structure that has two different dummy sections, one for 32-bit register CPU's and the other for 64-bit register CPU's is register dependent.

The alternative is using a structure that can accommodate either, meaning designed for 64-bit registers. The same displacement can be used for either register size, allowing a single dummy section to define the structure in all cases. The extra four bytes of the field are essentially unused when the structure is in use on a 32-bit register system. Such a structure is register independent.

The function frame structure is an example of a register dependent structure. It utilizes a different dummy section for each of the register dependent definitions. A standard prefix is used for each field in the frame's dummy section to differentiate the two cases.

The `IOCB` structure is an example of a register independent structure. The address fields used for the channel subsystem control blocks are all eight bytes in length. The same field names can be used regardless of the architecture, although the code that uses the structure must ensure the correct instruction is utilized for the respective architecture. Operation synonyms provide that consistency.

Generally, the program implementer must make a choice between the two approaches when supporting 32-bit register architectures along with z/Architecture (64-bit registers) in the same program.

In the two examples, SATK has chosen each approach in two different situations. The IOCB is used for a single device. Bare-metal programs typically do not utilize a large number of devices. The penalty for the extra four bytes on 32-bit systems is small. SATK has chosen the register independent approach for the `IOCB`.

In the case of function frames, the size of each frame drives the size of the stack. Multiple stacks are likely with multiple frames in each stack possible. The penalty for the excess area when applied to each saved register becomes much higher. SATK has opted to optimize for smaller size in the case of functions by using two different dummy sections depending upon register size.

This does mean that a function assembled for use on a 32-bit system **can not** be directly used on a system operating with 64-bit registers even if the code within the function can. Such a function must be, at a minimum assembled separately under architecture level 9 so that the function prolog and epilog code is consistent.

Channel Command Words

Channel Command Word formats have their own considerations when code needs to support both formats. While only one is valid for architecture levels 4 and below, 5 and above can

support both. At assembly time, the use of the XMODE CCW setting controls which is created by the CCW assembly directive. The XMODE CCW setting will also dictate the value assigned to the &SYSCCW system variable.

Code sensitive to CCW format, must examine the &SYSCCW system variable to determine which fields from the respective CCW dummy sections are required for the specific code being created.

At run-time, for code that is sensitive to the format and needs to support both, the ORB structure should be examined to determine the actual format in use and utilize the code tailored for each case.

Sharing Executable Code

ARCHIND macro operator synonyms help with sharing source code between different architectures. For code intended for a single architecture build, operator synonyms work well.

However when multiple architectures are involved, the source code must be assembled under an architecture level that ensures the assembled instructions will execute in different architectures correctly. Where general instructions are involved, this usually means assembling it under the lowest supported architecture level and sharing it.

This approach does not work where different control instructions are involved. The most general case where this applies is in the performance of input/output operations. Separate assembly of input/output operations are required for the two different architectures when both are supported by the same program. Architectures that do not share the same input/output architecture can not typically share the same input/output code even if other architectural attributes are the same, for example, System/370 in extended-control mode vs. System/370 Extended Architecture.

Code Sharing in Multiple Architectures

Traditional multiples uses in mainframe assembly of the same code is normally achieved through the use of a macro. Symbols within a mainframe assembly are global to the assembly. A symbol receives a value that does not change. To accommodate using the same code with symbols, the macro incorporates the &SYSNDX symbolic variable in its symbol creation. This allows reuse of the same macro while creating unique symbols within each usage. The &SYSNDX symbolic variable is incremented upon each new occurrence of a macro statement. Hence, its value in any specific macro is unpredictable. As such, the use of &SYSNDX does not work to address symbol creation where a predictable symbol is required.

Modifying the approach to utilize a symbolic variable that has a predictable value for a given context, though, does address the needs of code shared between multiple architectures, and not others. The approach is generic, but requires all source code expected to be shared in this way be placed within a macro. It is suggested that this occur through the use of functions. Functions provide for isolation of code tagged by a single symbol, namely the

function's symbol. By providing wrapper macros that know which symbolic variable to utilize for a given sharing of code, a single set of source code can be used to assemble functions used by multiple architectures.

Such symbolic variables are only required where different architectures can share specific the same code but not others. The compatibility drivers described above dictate where such symbolic variables are needed:

- structures with architecture sensitive field sizes (in particular functions themselves),
- input/output architecture compatibility and
- architecture specific

The `ARCHIND` macro facilitates the setting of the values for a given architecture. The following global character symbolic variables are used when defining symbols:

- `&A` – architecture specific
- `&I` – input/output architecture sensitive
- `&S` – function shared by multiple architectures

Needed granularity for specific architecture related code can be achieved within a macro by using tests on the `&ARCHLVL` global arithmetic symbolic variable to control code generation. In many cases the need for such is eliminated by operation synonyms. However, operation synonyms apply to individual instructions. In some contexts, unique instruction sequences are required in different contexts and the `&ARCHLVL` variable aids in correct code generation.

The following table illustrates how the scheme applies to each architecture level. Levels that can share code are shown with the same background color. No special meaning is given to the color itself. The `--target` column indicates the ASMA target recommended for use when supporting this architecture. A target of `24` is the default if no target has been supplied. The `&SYSCCW` column indicates the CCW format established by the target. The –psw column indicates the ASMA `--psw` value when assembling a program supporting this corresponding architecture. `XMODE PSW` column indicates the assembler setting that should be used when assembling code specific to this architecture. The `&ARCHLVL` column indicates the architecture level assigned by the `ARCHLVL` macro for the architecture. The three columns `&S`, `&I` and `&A` identify the values assigned to the symbolic variable intended to allow source code sharing between, respectively, code sensitive to register size, input/output architectures and a specific architecture.

Architecture	--target	&SYSCCW	--psw	XMODE PSW	&ARCHLVL	&S	&I	&A
S/360	24	0	360	360	1	'F'	'C'	'1'
S/370 BC	24	0	360	BC	2	'F'	'C'	'2'
S/370 EC	24	0	EC	EC	3	'F'	'C'	'3'
S/380	24	0	EC	380	4	'F'	'C'	'4'
370-XA	24	0	EC	XA	5	'F'	'D'	'5'

Architecture	--target	&SYSCCW	--psw	XMODE PSW	&ARCHLVL	&S	&I	&A
ESA/370	24	0	EC	E370	6	'F'	'D'	'6'
ESA/390	24	0	EC	E390	7 (Note 1)	'F'	'D'	'7'
ESA/390 on z	24	0	EC	E390	8	'F'	'D'	'8'
z/Architecture	24/64	0/1	EC	Z	9	'G'	'M'	'9'

Note 1: When assembling code specific to this architecture it must be preceded by an `ARCHLVL` macro with the `ZARCH=NO` parameter. Otherwise the assembly will assume the architecture is ESA/390 on a z/Architecture system.

What this table illustrates is that a single assembly, and by extension a single version of an IPL medium, can be utilized for all architectures except those that utilize the basic control mode format PSW. A current limitation of the `iplasma.py` utility requires a separate assembly and input list-directed IPL directory when the IPL PSW in the basic-control format is required.

Coding for Multiple Architectures

How to code for multiple architecture support involves proper structuring of the source code and utilizing the compatibility requirements and assists.

See the sample program `samples/asma/hellofm.asm` for an example of how a program should be structured. The `hellofm.asm` sample extends the `hellof.asm` sample for multiple architectures.

The assumption is made that the output format for the assembly is a list-directed IPL directory, the ASMA `-g` or `--gldipl` command line option.

The strategy involves assembling a version of the same source code under different architecture contexts. By “same source code” is meant either the same source file copied into the program multiple times or a macro invoked multiple times. In either case, the architecture context is established by an `XMODE PSW` directive followed by the `ARCHLVL` macro, and implicitly or explicitly the `ARCHIND` macro:

```
          XMODE PSW,XXX
          XMODE CCW      OPTIONAL
          ARCHLVL
*         shared source code
```

Code Sharing Wrapper Macros

Macros can take advantage of the symbolic variables for selectively generating code in different settings. This is not available to code copied into the assembly. This makes macro encapsulation a better ASMA option.

For each portion of the **source** code intended to be shared between:

- CPU's of the same register size,
- architectures using the same input/output architecture or
- specific architectures (through operator synonyms, for example)

the source code is embedded in a macro that is selectively invoked.

When encapsulating code within a macro, care should be maintained with regards to the control section into which the code is being assembled. `USING` directives within the shared code should have corresponding `DROP` directives, ensuring following code selects the correct base registers.

Because a function naturally encapsulates a portion of the code, they become easy targets for implementation within a code sharing wrapper macro. To facilitate this, wrapper `FUNCTION` and `CALL` macros are provided that ensure the correct suffix is applied to a function. These are `AFUN`, `IFUN` and `SFUN` for functions, and `ACALL`, `ICALL` and `SCALL` for the `CALL` macro. In each case, the function definition and calling wrapper macro utilizes the same parameters as the underlying `FUNCTION` and `CALL` macro.

The `CALLR` macro is register based and does not require a wrapper. However, the code that sets up the register used by a `CALLR` macro, may need ensure the correct suffix is utilized when the address is created.

Because a program supporting multiple architectures, is most likely to be performing the same activities in each of the supported architectures, the code that drives the application is also a candidate for being shared. A macro encapsulating the basic processing of the application is also a strong candidate for being shared. It can utilize the `&A` symbolic variable for the creation of unique symbols in each of its architecture contexts.

Incremental Development

A program supporting multiple architectures is really multiple programs. Developing the program for a single architecture and testing it is recommended. If the program was not originally structured for multiple architectures, restructure it for the use of wrapper macros, but still assemble and test it for the one running architecture. Then add sections to the program supporting multiple architectures invoking the shared code as appropriate.

This approach was used in changing the `hellof.asm` sample into the `hellofm.asm` sample with fair success. The `hellof.asm` sample had the advantage of already working in multiple architectures although it required a separate assembly for each.