

Stand Alone Tool Kit Common Macros

Table of Contents

Notices.....	2
Introduction.....	2
How to Use This Document.....	2
ANTR.....	3
APROB.....	5
ARCHIND.....	7
ARCHLVL.....	10
ASAIPL.....	12
ASALOAD.....	13
ASAREA.....	14
ASAZAREA.....	15
DSECTS.....	17
DWAIT.....	19
DWAITEND.....	21
ESA390.....	22
POINTER.....	23
PSWFMT.....	24
SIGCPU.....	25
SMMGR.....	27
SMMGRB.....	29
TRAP64.....	30
TRAP128.....	31
TRAPS.....	32
ZARCH.....	34
ZEROH.....	35
ZEROL.....	36
ZEROLH.....	37
ZEROLL.....	38
HOWTO be Architecture Sensitive.....	39
APROB.....	39
ANTR.....	40
ARCHIND.....	40
ARCHLVL.....	41
Assembly Time.....	41
ESA/390 Considerations.....	41
Overriding the XMODE PSW Setting.....	41
Run Time.....	42
HOWTO Define Structures.....	43
HOWTO Change Architecture Mode.....	44
Side-Effects of Architecture-Mode Changes.....	44
Side-Effects of Address-Mode Changes.....	45
HOWTO Terminate the Program.....	47
HOWTO Fly Without a Net.....	48

Stand Alone Tool Kit Common Macros

TRAPS.....	48
ASALOAD.....	49
ASAIPL.....	49

Copyright © 2017 Harold Grovesteen

See the file `doc/fdl-1.3.txt` for copying conditions.

Notices

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

Introduction

This document describes various facilities provided by Stand Alone Tool Kit (SATK) using A Small Mainframe Assembler (ASMA) for interaction with the system architecture and CPU.

All of the software described in this manual resides in the `maclib` directory of SATK. To make the macros described here available to the assembler, the SATK `maclib` directory must be defined in the ASMA `MACLIB` environment variable.

How to Use This Document

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

For information related to architecture levels, refer to either `UserGuide.odt` or `UserGuide.pdf`. The architecture level concept apply to nearly all macros.

Stand Alone Tool Kit Common Macros

ANTR

Source file: maclib/ANTR.mac

Macro Format:

```
[label]    ANTR  [REG=[1|n]]  
            [,S360=label]  
            [,S370BC=label]  
            [,S370=label]  
            [,S380=label]  
            [,XA=label]  
            [,E370=label]  
            [,E390=label]  
            [,S390=label]  
            [,S390X=label]
```

The `ANTR` passes control to the program location specified by its corresponding label for each recognized architecture level.

If the architecture level is not recognized, a disabled wait state occurs. The following table identifies the address of the disable wait PSW if the running architecture is not recognized by the program because the corresponding keyword is omitted or empty.

PSW Address	Keyword	Description
0200	none	Running architecture is unknown or invalid.
0201	S360	Program does not recognize or support use on a System/360
0202	S370BC	Program does not recognize or support use on System/370 in basic-control mode
0203	S370	Program does not recognize or support use on System/370 in extended-control mode
0204	S380	Program does not recognize or support use on Hercules System/380
0205	XA	Program does not recognize or support use on System 370 Extended Architecture system
0206	E370	Program does not recognize or support use on ESA/370 system
0207	E390	Program does not recognize or support use on a native ESA/390 system
0208	S390	Program does not recognize or support use on a z/Architecture system in ESA/390 mode
0209	S390X	Program does not recognize or support direct entry in native z/Architecture mode

Stand Alone Tool Kit Common Macros

Assembly Considerations:

- A base register other than the register identified by the `REG` keyword parameter must be established prior to assembling the macro.
- `ANTR` is independent of the `ARCHLVL` or `ARCHIND` macros and may occur before either is used by the assembly.

Execution Considerations:

- The contents of the register identified by the `REG` keyword parameter must be a positive integer between 1 and 9 corresponding to the running architecture. The `APROB` macro can detect the value for the running system.

Label Field Usage:

The label field is associated with the first instruction generated by the macro when present.

Positional Parameters: None

Keyword Parameters:

Keyword	Default	Description
E370	none	The label to which control is passed if the running architecture is ESA/370.
E390	none	The label to which control is passed if the running architecture is a ESA/390 not on z/Architecture.
REG	1	Identifies the register containing the running architecture level.
S360	none	The label to which control is passed if the running architecture is S/360.
S370	none	The label to which control is passed if the running architecture is S/370 in extended control mode.
S370BC	none	The label to which control is passed if the running architecture is S/370 in basic control mode.
S380	none	The label to which control is passed if the running architecture is Hercules S/380.
S390	none	The label to which control is passed if the running architecture is ESA/390 on a z/Architecture system.
S390X	none	The label to which control is passed if the running architecture is z/Architecture.
XA	none	The label to which control is passed if the running architecture is 370/XA.

Programming Note:

The value expected in the register identified by the `REG` keyword parameter is anticipated to be supplied by the `APROB` macro's execution.

A program might recognize but not support use of a specific architecture if it expects to provide more user friendly error reporting than is available with a disable wait state.

Each location to which control is passed and the program supports the architecture, the label will likely be preceded by its own `ARCHLVL` or `ARCHIND` macro.

Stand Alone Tool Kit Common Macros

APROB

Source file: maclib/APROB.mac

Macro Format:

```
[label]    APROB [HERC=[YES|NO]]  
           [,S360=[YES|NO]]  
           [,S380=[YES|NO]]  
           [,REG=1|n]
```

The `APROB` macro detects the executing architecture and determines its architecture level. The architecture level, a positive integer between 1 and 9, inclusive, is placed in the specified register.

The keyword arguments control whether detection is generated for an architecture. If `APROB` is executed on a system for which detection has not been generated, the next higher architecture level that is detected will be reported. The following table identifies the reported value for each detectable architecture.

Architecture	Level	Required for Detection
System/360 (S/360)	1	HERC=NO, S360=YES
System/370 (S/370) basic-control mode	2	Always detected
System/370 extended-control mode	3	Always detected
Hercules System/380 (S/380)	4	S380=YES
System/370 Extended Architecture (370/XA)	5	HERC=NO
Enterprise System Architecture/370 (ESA/370)	6	HERC=NO
Enterprise System Architecture/390 (ESA/390)	7	Always detected
Enterprise System Architecture/390 on z/Architecture	8	Always detected
z/Architecture	9	Always detected

A value of 0 indicates an error condition occurred and architecture detection failed. While this is unlikely, it indicates a failure of the detection processes.

Assembly Considerations:

- A base register other than the register identified by the `REG` keyword parameter must be established prior to assembling the macro.
- `APROB` is independent of the `ARCHLVL` or `ARCHIND` macros and may occur before either is used by the assembly.
- `APROB` can be assembled regardless of the targeted assembly architecture. Instructions that may not be available are coded as hexadecimal constants.

Stand Alone Tool Kit Common Macros

Execution Considerations:

- Program and Supervisor Call New PSW's are used to perform the architecture detection. Contents prior to executing `APROB` of all new PSW assigned storage areas are restored upon completion of the detection process.

Label Field Usage:

The label field is associated with the first instruction generated by the macro when present.

Positional Parameters: None

Keyword Parameters:

Keyword	Default	Description
HERC	NO	Specify YES to detect only Hercules standard architectures: <ul style="list-style-type: none">• S/370 in basic control mode (architecture level 2),• S/370 in extended control mode (architecture level 3),• ESA/390 (architecture level 8),• ESA/390 on a z/Architecture system (architecture level 9), and• z/Architecture (architecture level 9).
S360	NO	Specify YES to detect S/360 (architecture level 1). If NO is specified or HERC=YES, S/360 will be detected as S/370 in basic-control mode (architecture level 2). Only specify YES if an S/360 system or compatible is running. For compatibility purposes S/370 in basic-control mode is not compatible.
S380	NO	Specify YES to detect Hercules S/380 (architecture level 4). If NO is specified, S/380 will be detected as 370/XA (architecture level 5) when HERC=NO, or as ESA/390 (architecture level 8) when HERC=YES. Only specify YES when running on a Hercules S/380 system.
REG	1	The register into which the detected architecture level is placed. Must not be register 1, 2, or 3 if S380=YES.

Programming Note:

Reported values may be used with the `ANTR` macro to pass control to a specific portion of the program designed to support the detected architecture. If `ANTR` is used, the `REG` keyword argument may be the same register as used by `APROB`.

Stand Alone Tool Kit Common Macros

ARCHIND

Source file: `maclib/ARCHIND.mac`

Macro Format:

`ARCHIND`

The `ARCHIND` macro provides

- architecture independent operation synonyms for various instructions depending upon the current assembly architecture level, and
- architecture sharing function name suffixes.

By using the operation synonym a program can cause the correct instruction for the architecture to be assembled frequently allowing code reuse between different architecture.

By using function sharing wrapper macros, functions of the same name in source code can be transparently shared by different architectures. The architecture specific functions use the transparently appended function name suffix to differentiate between the functional equivalent functions. The suffix is assigned by the preceding `ARCHIND` macro.

The following table indicates the synonyms and the actual instruction for all architecture levels. '--' indicates no form of the instruction is supported by the architecture.

The column for architecture level 0 are the synonyms established when the `ARCHLVL` macro has not established the current assembly architecture. The first three rows identify the assigned function differentiating suffix for shared (S), input/output (I) or architecture explicit (A) functions.

Synonym	0	1	2-4	5, 6	7, 8	9
S	' '	'F'	'F'	'F'	'F'	'G'
I	' '	'C'	'C'	'D'	'D'	'M'
A	' '	'1'	'2', '3', '4'	'5', '6'	'7', '8'	'9'
\$AHI	--	--	--	--	AHI	AGHI
\$AL	AL	AL	AL	AL	AL	ALG
\$ALR	ALR	ALR	ALR	ALR	ALR	ALGR
\$B	B	B	B	B	J	J
\$BAS	BAS	BAL	BAS	BAS	BAS	BAS
\$BASR	BASR	BALR	BASR	BASR	BASR	BASR
\$BC	BC	BC	BC	BC	BRC	BRC
\$BCTR	BCTR	BCTR	BCTR	BCTR	BCTR	BCTGR

Stand Alone Tool Kit Common Macros

Synonym	0	1	2-4	5, 6	7, 8	9
\$BE	BE	BE	BE	BE	JE	JE
\$BH	BH	BH	BH	BH	JH	JH
\$BL	BL	BL	BL	BL	JL	JL
\$BM	BM	BM	BM	BM	JJM	JM
\$BNE	BNE	BNE	BNE	BNE	JNE	JNE
\$BNH	BNH	BNH	BNH	BNH	JNH	JNH
\$BNL	BNL	BNL	BNL	BNL	JNL	JNL
\$BNM	BNM	BNM	BNM	BNM	JNM	JNM
\$BNO	BNO	BNO	BNO	BNO	JNO	JNO
\$BNP	BNP	BNP	BNP	BNP	JNP	JNP
\$BNZ	BNZ	BNZ	BNZ	BNZ	JNZ	JNZ
\$BO	BO	BO	BO	BO	JO	JO
\$BP	BP	BP	BP	BP	JP	JP
\$BXLE	BXLE	BXLE	BXLE	BXLE	JXLE	JXLEG
\$BZ	BZ	BZ	BZ	BZ	JZ	JZ
\$CH	CH	CH	CH	CH	CH	CGH
\$CHI	--	--	--	--	CHI	CGHI
\$L	L	L	L	L	L	LG
\$LH	LH	LH	LH	LH	LH	LGH
\$LHI	--	--	--	--	LHI	LGHI
\$LPSW	LPSW	LPSW	LPSW	LPSW	LPSW	LPSWE
\$LR	LR	LR	LR	LR	LR	LGR
\$LTR	LTR	LTR	LTR	LTR	LTR	LTGR
\$NR	NR	NR	NR	NR	NR	NGR
\$SL	SL	SL	SL	SL	SL	SLG
\$SLR	SLR	SLR	SLR	SLR	SLR	SLGR
\$SR	SR	SR	SR	SR	SR	SGR
\$ST	ST	ST	ST	ST	ST	STG
\$STM	STM	STM	STM	STM	STM	STMG
\$X	X	X	X	X	X	XG

Stand Alone Tool Kit Common Macros

Assembly Considerations:

- The assembler's `--target` command-line option controls instruction availability. An attempt to establish an operation synonym for an unavailable instruction results in an assembly error. The assembly should target the highest architecture level expected when multiple architecture levels are being assembled. Use of `--target 24`, `--target 31`, or `--target 64` are recommended.

Execution Considerations:

- The run-time architecture level must match the assembly architecture level. Otherwise program operation exceptions can occur.

Label Field Usage: Prohibited

Positional Parameters: None

Keyword Parameters: None

Programming Note:

Macros `APROB` and `ANTR` allow a program to ensure its run-time architecture matches the architecture targeted by the assembly-time `ARCHLVL` macro.

Stand Alone Tool Kit Common Macros

ARCHLVL

Source file: `maclib/ARCHLVL.mac`

Macro Format:

```
[label]  ARCHLVL [ZARCH=[YES|NO]]  
          [,PSW=xxx]  
          [,SET=n]  
          [,ARCHIND=[YES|NO]]  
          [,MNOTE=[YES|NO]]
```

The `ARCHLVL` macro interrogates the assembly environment determining the architecture level of the program being assembled. Unless indicated otherwise, the current `XMODE` PSW setting initialized by the assembly architecture target (specified by the `--target` command-line option) is examined to determine the architecture level. By default

The global arithmetic symbolic variable `&ARCHLVL` is set based upon the identified architecture level. Macros dependent upon the assembly architecture level should use the value in this symbolic variable and test for a value of zero, indicating `ARCHLVL` was not previously invoked in the assembly, for validity.

The priority of sources, with the highest priority first, are:

1. `PSW` keyword parameter overrides,
2. `SET` keyword parameter overrides,
3. `--target` command-line option.

The following table identifies the architecture level depending upon the interrogated source.

PSW=	SET=	--target	ZARCH=	Assembly Architecture
360	1	s360	ignored	System/360 (S/360)
BC	2		ignored	System/370 (S/370) in basic-control mode
EC	3	s370 or 24	ignored	System/370 in extended control mode
380	4	s380	ignored	Hercules System/380 (S/380)
XA	5	370xa	ignored	System/380 Extended Architecture (370-XA)
E370	6	e370	ignored	Enterprise System Architecture/370 (ESA/370)
E390	7	E390 or 31 or 64	NO	Enterprise System Architecture/390 (ESA/390)
S390	8	s390 or 31 or 64	YES	ESA/390 on z/Architecture
Z	9	s390x	ignored	z/Architecture

Assembly Considerations: None

Stand Alone Tool Kit Common Macros

Execution Considerations: None

Label Field Usage:

If provided, the label field's symbol is equated to the identified assembly architecture level.

Positional Parameters: None

Keyword Parameters:

Keyword	Default	Description
ARCHIND	YES	Specify YES to have ARCHLVL automatically call ARCHIND after the assembly architecture level is determined.
MNOTE	YES	Specify NO to inhibit the informational MNOTE containing the value to which &ARCHLVL has been set.
PSW	none	Force the current XMODE PSW setting to this value before detecting the assembly architecture level.
SET	none	Force the assembly architecture to the specified value between 1 and 9, inclusive, without altering the current XMODE PSW setting. PSW takes precedence over this value.
ZARCH	YES	Indicates whether ESA/390 is on a z/Architecture system (YES) or not (NO). Otherwise ignored

Programming Note:

System/370 in basic-control mode can only be set by using PSW=BC or SET=2.

The assembly architecture level may be communicated to the running program by assembling an A or AD address constant with the ARCHLVL macro's label. The value of the constant can be compared to the value provided by the run-time APROB macro to determine if the run-time architecture matches the architecture expected by the assembly. By simply comparing for equality the APROB register contents to the assembled address constant will validate if they match. At run-time, the ANTR macro will do precisely that based upon its assembled key-word arguments.

Stand Alone Tool Kit Common Macros

ASAIPL

Source file: `maclib/ASAIPL.mac`

Macro Format:

```
[label]   ASAIPL  [IMSK=[0|n]]  
           [ ,KEY=[0|n]]  
           [ ,SYS=[0|n]]  
           [ ,PGM=[0|n]]  
           ,IA=entry  
           [ ,AM=[24|31|64]]
```

The `ASAIPL` macro assembles an IPL PSW at the start of the assigned storage area control section. The PSW format is determined by the current assembly architecture level.

The IPL PSW overlays the restart new PSW provided by the `ASALOAD` macro.

Assembly Considerations:

- The `ASAIPL` macro must be preceded in the assembly by a `ASALOAD` macro creating the assigned storage area control section.
- One of the PSW assembler directives is used to create the PSW. All values allowed by the corresponding PSW directive operands are valid. Refer to `ASMA.odt` or `ASMA.pdf` for details related to these directives' operands.

Execution Considerations:

- The IPL PSW must reside in absolute addresses 0-7. The technique used to load the assembler output into system memory must ensure that happens.

Label Field Usage:

If provided, the label field's symbol is associated with the IPL PSW within the assigned storage area control section.

Positional Parameters: None

Keyword Parameters:

Keyword	Default	Description
AM	24	Specifies the address mode of the IPL PSW. Valid values are 24, 31, or 64.
IA	none	Specifies the IPL PSW instruction address, the location to which control is passed during the IPL function. This is the entry point of the program. It is required.
IMSK	0	Specifies the IPL PSW interruption mask.
KEY	0	Specifies the IPL PSW storage key.
PGM	0	Specifies the IPL PSW program mask.
SYS	0	Specifies the IPL PSW system mask.

Stand Alone Tool Kit Common Macros

ASALOAD

Source file: maclib/ASALOAD.mac

Macro Format:

```
[label] ASALOAD [REGION=ASAREGN]
           [ , ZARCH=[YES | LEVEL | NO] ]
           [ , PSW=psw]
```

The `ASALOAD` macro creates an assigned storage area control section in the specified region. Interruption trap PSW's are placed within the control section at their assigned storage locations.

Assembly Considerations:

- `ASALOAD` depends upon `ARCHLVL` or `ARCHIND` if `PSW` keyword parameter is omitted or `ZARCH=LEVEL` is specified.

Execution Considerations:

- Failure to follow `ASAIPL` by `ASALOAD` will result in the restart new trap PSW being loaded by the IPL function, resulting in a disabled wait state.

Label Field Usage:

If provided, the label field's symbol is the name of the assigned storage area control section. Omitting the label results in the assigned storage area control section being the unnamed control section of the assembly.

Positional Parameters: None

Keyword Parameters:

Keyword	Default	Description
PSW	none	Force usage of this PSW format for 64-bit interrupt trap PSW's. If omitted defaults to the PSW format of the current assembly architecture level.
REGION	ASAREGN	The region into which the assigned storage area control section is placed as its first control section.
ZARCH	LEVEL	Specifies whether 128-bit interrupt trap PSW's are to be placed into the assigned storage area control section. Specify: <ul style="list-style-type: none">• <code>YES</code> to force inclusion of 128-bit interrupt trap PSW's• <code>NO</code> to inhibit inclusion of any 128-bit interrupt trap PSW's• <code>LEVEL</code> to only include 128-bit interrupt trap PSW's if the current assembly architecture level would use them, namely, for architecture levels 8 or 9.

Stand Alone Tool Kit Common Macros

ASAREA

Source file: maclib/ASAREA.mac

Macro Format:

```
[label] ASAREA [DSECT=[YES|NO]]  
[ , SCANOUT=[0|n]]
```

The `ASAREA` macro provides definitions of all assigned storage areas for all architecture levels within the first 256 bytes of memory.

The definitions may be placed in an existing control or dummy section or within a new one.

Assembly Considerations: None

Execution Considerations: None

Label Field Usage:

When the label field is present, the definitions are placed in a new control section as indicated by the `DSECT` keyword parameter. If omitted the definitions are placed at the current location the current active section.

Positional Parameters: None

Keyword Parameters:

Keyword	Default	Description
DSECT	NO	Controls the type of new section into which definitions are placed when the label field is present: <ul style="list-style-type: none">• <code>NO</code> creates a new control section• <code>YES</code> create a new dummy section.
SCANOUT	0	Some System/360 systems assign scan out area. This value indicates its size in bytes.

Stand Alone Tool Kit Common Macros

ASAZAREA

Source file: `maclib/ASAZAREA.mac`

Macro Format:

```
[label] ASAZAREA [DSECT=[YES|NO]]  
                [,ORG=[YES|n]]
```

The `ASAZAREA` provides definitions of save areas used by z/Architecture systems not contiguous with the areas defined by `ASAREA`. The `ASAZAREA` definitions may be placed in a section of it own or within the same section as `ASAREA`.

Assembly Considerations: None

Execution Considerations: None

Label Field Usage:

When the label field is present, the definitions are placed in a new control section as indicated by the `DSECT` keyword parameter and as adjusted by the `ORG` keyword parameter is supplied.. If omitted, the definitions are placed at the current location the current active section.

Positional Parameters: None

Keyword Parameters:

Keyword	Default	Description
DSECT	NO	Controls the type of section into which definitions are placed when the label field is present: <ul style="list-style-type: none">• <code>NO</code> creates a new control section• <code>YES</code> creates a new dummy section. The section may be a new section or continuation of an existing section.
ORG	none	Positions the definitions within the section identified by the label. <ul style="list-style-type: none">• <code>YES</code> positions the definitions at a location of X'11C0' bytes from the start of the label field identified section.• Another value will position the definitions at a location beyond the start of the label field identified section by the number of bytes supplied by the supplied value. Ignored if the <code>label</code> field is not present.

Programming Note:

Access to the fields requires a base register. Whether the same or different section is used, is driven by the type of instructions used to access the fields. 20-bit displacement instructions allow direct access with base register 0 and can be reasonably included in the same section defining the `ASAREA` fields. Other instructions will require a base register other than 0 and would most likely be placed in a different section.

Stand Alone Tool Kit Common Macros

The following sequence will place both areas in the same dummy section:

```
ASADSECT ASAREA DSECT=YES
ASADSECT ASAZAREA DSECT=YES,ORG=YES
        USING ASADSECT,0
```

In this case both could be accessed using register 0 and the second area using 20-bit displacement instructions.

This sequence will place the two areas in different dummy sections, using different base registers for access to the two areas:

```
ASA      ASAREA DSECT=YES
ASAZ      ASAZAREA DSECT=YES
        USING ASA,0
        L      2,=A(X'11C0')
        USING ASAZ,2
```


Stand Alone Tool Kit Common Macros

DSECTS

Source file: `maclib/DSECTS.mac`

Macro Format:

```
[label] DSECTS [PRINT=[ON|OFF]]  
            [,NAME=[name| (name1,name2,...)]]
```

The `DSECTS` macro is a convenience macro for generating SATK supplied dummy section structure definitions.

The following table summarizes the selected structures. Names that include “or” will be selected by name or the current assembly architecture level if no name is supplied. Names without “or” are only defined when the name is present, but the current assembly architecture level dictates the specific section definition included in the assembly.

NAME	Level	Manual	Description
ASA	--	SATK	Defines assigned storage areas
ASAZ	--	SATK	Defines z/Architecture save areas
CCW	--	sync	Defines both CCW formats
CCW0	--	sync	Defines the Format-0 Channel Command Word
CCW1	--	sync	Defines the Format-1 Channel Command Word
CHAN or	1-4	sync	Defines structures used by channel-based I/O: CCW0 and CSW
CS or	5-9	sync	Defines structures used by channel subsystem-based I/O: CCW0, CCW1, IRB, ORB, SCHIB and SCSW
CSW	--	sync	Defines the Channel Status Word (CSW)
FRAME	1-8	func	Defines the 32-bit register function stack frame (DSECT <code>STKF</code>)
	9		Defines the 64-bit register function stack frame (DSECT <code>STKG</code>)
IO	--	sync	Defines all I/O related structures
IOCB	--	sync	Defines the raw I/O control block
IRB	--	sync	Defines the Interruption-Response Block
ORB	--	sync	Defines the Operation-Request Block
PSW or	1-4	SATK	Defines the 64-bit basic-control mode PSW (DSECT <code>PSWB</code>)
	2-9		Defines the 64-bit extended-control mode PSW (DSECT <code>PSWE</code>)
	9		Defines the 128-bit z/Architecture PSW (DSECT <code>PSWZ</code>)
SCHIB	--	sync	Defines the Subchannel Information Block
SCSW	--	sync	Defines the Subchannel Status Word
TABLE	--	table	Defines the SATK table definition structures (DSECT <code>TBL</code> and <code>TBLG</code>)

Stand Alone Tool Kit Common Macros

No structure is defined more than once.

Assembly Considerations:

- `DSECTS` requires a preceding `ARCHLVL` macro.

Execution Considerations: None

Label Field Usage: Prohibited

Positional Parameters: None

Keyword Parameters:

Keyword	Default	Description
NAME	none	A name or a sublist of names of those structures being defined. If omitted the current assembly architecture level influences the PSW format(s) and I/O related structures defined. See the preceding table for details.
PRINT	none	Specifies how the assembly <code>PRINT</code> controls are to be set during structure definition. When specified, the <code>PRINT</code> controls are restored following structure definition. If omitted, the <code>PRINT</code> controls in effect when the <code>DSECTS</code> macro is invoked are used unchanged.

Stand Alone Tool Kit Common Macros

DWAIT

Source file: `maclib/DWAIT.mac`

Macro Format:

```
[label]   DWAIT  [PGM=[01|pp]]
           [,CMP=[0|c]]
           [,CODE=[BAD|rrr]]
           [,LOAD=[YES|NO]]
```

The `DWAIT` macro creates a disabled wait state PSW with an instruction address that identifies the program, its component, and a reason code for the program's abnormal termination.

The information is conveyed in the following hexadecimal formatted field placed in the disabled wait PSW's instruction address field:

`ppcrrr`

where:

- `pp` is the program's hexadecimal identifier, `00` reserved for SATK;
- `c` is the program's component hexadecimal identifier; and
- `rrr` is the hexadecimal identifier of the reason the disabled wait state condition was entered.

If no identifiers are supplied the instruction address defaults to `X'010BAD'`.

The assembled PSW is that of the current assembly architecture level. The reporting format is compatible with all architectures.

Assembly Considerations:

- `DWAIT` requires a preceding `ARCHIND` macro in the assembly.
- A base register is required if `LOAD=YES` is used.

Execution Considerations: None

Label Field Usage:

The label field is associated with the first instruction generated by the macro when present.

Positional Parameters: None

Keyword Parameters:

Keyword	Default	Description
CMP	0	The program component, the <code>c</code> hexadecimal digit of the instruction address.

Stand Alone Tool Kit Common Macros

Keyword	Default	Description
CODE	BAD	The reason code for why the program entered the disabled wait state, the <code>rrr</code> hexadecimal digits of the instruction address.
LOAD	NO	Whether the disabled wait state should be loaded as the current PSW <code>LOAD=YES</code> or just assembled <code>LOAD=NO</code> .
PGM	01	The program identifier, the <code>pp</code> hexadecimal digits of the instruction address. Program identifier <code>00</code> is reserved for use by SATK.

Programming Note:

Programmers are encouraged to document the usage of the disabled wait states whether created by `DWAIT` or a different reporting scheme for the instruction address field.

Stand Alone Tool Kit Common Macros

DWAITEND

Source file: `maclib/DWAITEND.mac`

Macro Format:

```
[label] DWAITEND [LOAD=[YES|NO]]
```

The `DWAITEND` macro creates a disabled wait PSW indicating the normal termination of the program. The instruction address field of the PSW is set to zero.

The assembled PSW is that of the current assembly architecture level.

Assembly Considerations:

- `DWAITEND` requires a preceding `ARCHIND` macro in the assembly.
- A base register is required if `LOAD=YES` is used.

Execution Considerations: None

Label Field Usage:

The label field is associated with the first instruction generated by the macro when present.

Positional Parameters: None

Keyword Parameters:

Keyword	Default	Description
LOAD	NO	Whether the disabled wait state should be loaded as the current PSW <code>LOAD=YES</code> or just assembled <code>LOAD=NO</code> .

Programming Note:

It is recommended that `DWAITEND` be used when normally terminating a stand alone program.

Stand Alone Tool Kit Common Macros

ESA390

Source file: `maclib/ESA390.mac`

Macro Format:

```
[label]  ESA390 pair, cpur
          [, SUCCESS=label]
          [, FAIL=label]
```

The `ESA390` macro changes the system architecture from z/Architecture to ESA/390.

Assembly Considerations:

- `ESA390` requires a preceding `ARCHLVL` macro in the assembly.

Execution Considerations:

- See the section “Side-Effects of Architecture Mode Changes” for execution considerations when using `ESA390` macro.
- The condition code is set with the same conditions as described by the `SIGCPU` macro.

Label Field Usage:

The label field is associated with the first instruction generated by the macro when present.

Positional Parameters:

1. The `pair` positional parameter requires the identification of an even/odd general register pair used to the architecture change.
2. The `cpur` positional parameter requires the identification of another general register used by the architecture change.

Keyword Parameters:

Keyword	Default	Description
FAIL	none	The label to which control is passed upon successful change of the architecture. If omitted, control passes to the next sequential instruction.
SUCCESS	none	The label to which control is passed upon successful change of the architecture. If omitted, control passes to the next sequential instruction.

Programming Note:

If neither `FAIL` nor `SUCCESS` keyword parameters are supplied, control passes to the next sequential instruction regardless of the success or failure of the architecture change. The condition code can be tested by the program for success or failure and the reasons.

Stand Alone Tool Kit Common Macros

POINTER

Source file: `maclib/POINTER.mac`

Macro Format:

```
[label]  POINTER loc
```

The `POINTER` macro creates an address constant of the correct size for the current assembly architecture level.

Assembly Considerations:

- `POINTER` requires a preceding `ARCHLVL` or `ARCHIND` macro in the assembly.

Execution Considerations: None

Label Field Usage:

The label field is associated with the created address constant when present.

Positional Parameters:

1. The required `loc` positional parameter identifies the location for which the address constant is being created.

Keyword Parameters: None

Stand Alone Tool Kit Common Macros

PSWFMT

Source file: `maclib/PSWFMT.mac`

Macro Format:

`PSWFMT`

The `PSWFMT` macro creates one or more dummy sections for the PSW formats encountered by the current assembly architecture level. The following dummy sections may be assembled.

Dummy Section	Level	Size (bits)	Program Status Word Format
PSWB	1-4	64	Basic-control mode
PSWE	2-9	64	Extended-control mode
PSWZ	8, 9	128	z/Architecture

Assembly Considerations:

- `PSWFMT` requires a preceding `ARCHLVL` or `ARCHIND` macro in the assembly.

Execution Considerations: None

Label Field Usage: Prohibited

Positional Parameters: None

Keyword Parameters: None

Stand Alone Tool Kit Common Macros

SIGCPU

Source file: `maclib/SIGCPU.mac`

Macro Format:

```
[label]    SIGCPU pair, cpur,  
           [, ORDER=[0|n]]  
           [, CPUADDR=label]  
           [, SUCCESS=label]  
           [, FAIL=label]
```

The `SIGCPU` macro issues a SIGNAL PROCESSOR instruction order to the executing CPU or some other CPU.

Following the execution of `SIGCPU`, the condition code will be set and control will pass to

- 0 – SUCCESS path is taken
- 1 – FAIL path is taken and the even register of the pair will contain detected conditions
- 2 – FAIL path is taken because the addressed CPU is busy
- 3 – FAIL path is taken because the addressed CPU is not operational.

Depending upon how the `SUCCESS` and/or `FAIL` keyword parameters are specified, the path may be to a label or the next sequential instruction following the macro.

Assembly Considerations:

- `SIGCPU` requires a preceding `ARCHLVL` macro in the assembly.
- The assembly architecture level must be greater than 1.
- A base register must be established prior to invoking `SIGCPU`.

Execution Considerations:

- The macro must be executed in privilege state.
- For orders that require a parameter, it must be placed in the odd register of the even/odd pair prior to executing the `SIGCPU` macro.

Label Field Usage:

The label field is associated with the first instruction of the macro when present.

Positional Parameters:

1. The `pair` positional parameter requires the identification of an even/odd general register pair used by the SIGNAL PROCESSOR instruction. The odd register of the pair contains the parameter information for orders that require it. The even register of the pair contains detected status conditions if the instruction failed.

Stand Alone Tool Kit Common Macros

2. The `cpur` positional parameter requires the identification of another general register used by the SIGNAL PROCESSOR instruction for specifying the address of the CPU to which the signal is presented.

Keyword Parameters:

Keyword	Default	Description
CPUADDR	none	The label of a halfword field containing the CPU address to which the signal is sent. If omitted, the executing CPU is assumed to be the target and the macro provides a field for saving its CPU address.
FAIL	none	The label to which control is passed upon successful change of the architecture. If omitted, control passes to the next sequential instruction.
ORDER	0	The SIGNAL PROCESSOR order being issued to the targeted CPU coded as a self defining term or label assigned to an absolute value
SUCCESS	none	The label to which control is passed upon successful change of the architecture. If omitted, control passes to the next sequential instruction.

Programming Note:

If neither `FAIL` nor `SUCCESS` keyword parameters are supplied, control passes to the next sequential instruction regardless of the success or failure of the architecture change. The program should directly test the condition code following SIGCPU execution to determine the results of the attempt.

Allowing the `ORDER` keyword parameter to default results in a failure of the SIGNAL PROCESSOR instruction because an order of 0 is unassigned.

Stand Alone Tool Kit Common Macros

SMMGR

Source file: `maclib/SMMGR.mac`

Macro Format:

```
[label]    SMMGR pair,pairw,w  
           [,BLOCK=[9|r]]  
           [,RETURN=r]
```

The `SMMGR` macro allocates memory statically. The allocations are permanent and made from lower memory addresses to higher memory addresses. The processing is inline and may be placed within a routine, subroutine or function.

The following table summarizes the register contents:

Register	On Macro Entry	Upon Macro Exit
<code>pair even</code>	Requested allocation length in bytes	Address of the aligned and allocated area
<code>pair odd</code>	Requested alignment factor or zero	unpredictable
<code>pairw even</code>	unknown	unpredictable
<code>pairw odd</code>	unknown	Address following the allocated area
<code>w</code>	unknown	Requested allocation size
<code>BLOCK</code>	<code>SMMGRB</code> address	<code>SMMGRB</code> address

Assembly Considerations:

- `SMMGR` requires a preceding `ARCHLVL` and `ARCHIND` macro in the assembly.
- In assembly architecture levels 1-6, a base register must be established before invoking `SMMGR`.

Execution Considerations:

- `SMMGR` only supports serial reuse when one `SMMGRB` is used. `SMMGR` is re-entrant if each CPU allocates using a different `SMMGRB`.
- The same instance of `SMMGR` may be used with the different `SMMGRB` managed areas by providing a different `SMMGRB` address in the `SMMGR BLOCK` register.
- The contents of the work registers are unpredictable upon completion of the allocation.

Label Field Usage:

The label field is associated with the first instruction of the macro when present.

Positional Parameters:

1. The required `pair` positional parameter identifies an even/odd register pair that

Stand Alone Tool Kit Common Macros

provides the allocation request to the allocation algorithm.

The even register contains the size of the area being requested. The odd register contains the alignment factor for the requested allocation. An allocation size of 0 aligns the next allocation to the requested alignment. Alignment must be in powers of two: 1, 2, 4, 8, etc. Alignments less than 1 are ignored.

Upon completion of the allocation, the even register of the `pair` positional parameter contains the starting address of the allocated area.

2. The required `pairw` positional parameter is a different even/odd pair of registers used by the allocation algorithm.
3. The required `w` positional parameter is another register used by the allocation algorithm.

Keyword Parameters:

Keyword	Default	Description
BLOCK	9	The input register that points to the address of the Static Memory Manager Block. See macro <code>SMMGRB</code> .
RETURN	none	If present, the specified register contains the address to which control is returned upon completion of the allocation. If omitted, control passes the next sequential instruction following <code>SMMGR</code> .

Programming Note:

The `SMMGR` macro may be used in-line, within a routine, a legacy subroutine or a function. For use within a function, the following is consistent with function calling conventions:

```
SMMGR 2, 6, 8
```

The function would need to supply the address of the Static Memory Manager Block in register 9 before invoking `SMMGR`.

Gaps created by different alignment requirements are not recovered. Allocations with large alignment factors, for example, 4096 for page alignment, are recommended before small allocations when feasible.

Stand Alone Tool Kit Common Macros

SMMGRB

Source file: `maclib/SMMGRB.mac`

Macro Format:

```
[label]    SMMGRB start
```

The `SMMGRB` macro creates the control block used by the Static Memory Manager allocation algorithm.

The block contains two 4- or 8-byte fields depending upon the assembly architecture level.

The first field identifies where the next allocation will be made before alignment occurs. As areas are allocated this field's content is increased towards higher addresses.

The second field is a mask used by the allocation algorithm in the performance of alignments. It is a field of all one bits, or a signed binary integer of -1.

Assembly Considerations:

- `SMMGRB` requires a `ARCHIND` macro in the assembly.
- Because the allocation algorithm requires access to `SMMGRB`, when placing the algorithm within a routine, subroutine or function, it is recommended the it be placed close to the algorithm making it easy to establish the `SMMGR BLOCK` register.

Execution Considerations:

- The register identified by the `SMMGR BLOCK` keyword parameter must contain the address of this macro.
- Multiple instances of `SMMGRB` may exist when multiple areas are being allocated with the Static Memory Manager algorithm.
- No boundaries are placed upon the area out of which `SMMGR` allocation requests are satisfied.

Label Field Usage:

The label field is associated with the start of the Static Memory Manager Block, if present.

Positional Parameters:

1. The required `start` positional parameter is a label or absolute address identifying the low water mark from which the Static Memory Manager allocates requested areas.

Keyword Parameters: None

Stand Alone Tool Kit Common Macros

TRAP64

Source file: maclib/TRAP64.mac

Macro Format:

```
[label]    TRAP64 [RESTART=[NO|ONLY|YES]]  
            [,PSW=[360|67|BC|EC|380|XA|E370|E390|Z]]
```

The `TRAP64` macro creates 64-bit disabled wait PSW's intended for placement at New PSW assigned storage locations. These PSW's allow the program to cease operation when an interruption class occurs for which no interrupt service routine is available.

Each interruption service routine trap PSW contains in its instruction address field the real location of the corresponding old PSW's assigned storage location.

Assembly Considerations:

- Each disabled wait PSW, except for the Restart trap PSW, is assembled in the same order in which their respective assigned storage new PSW's occur in memory.

Execution Considerations:

- If not assembled at the new PSW assigned storage location, the program must move the PSW's to their respective locations before the “trap” is enabled.

Label Field Usage:

The label field is associated with the first assembled PSW.

Positional Parameters: None

Keyword Parameters:

Keyword	Default	Description
PSW	none	When specified, overrides the current assembly <code>XMODE PSW</code> setting.
RESTART	NO	Specifies how a restart new “trap” PSW should be created: <ul style="list-style-type: none"><code>NO</code> indicates a new Restart PSW will not be created<code>ONLY</code> indicates that only a new Restart PSW is created. Others are not<code>YES</code> indicates the new Restart PSW will be created along with all of the others

Programming Note:

A “trap” Restart New PSW must not be assembled into its assigned storage location because an IPL PSW resides in the same memory location. In this case, a `TRAP64 RESTART=ONLY` should be used in the program, and following the IPL, the program must move the “trap” Restart New PSW to its assigned storage location at real address 0.

Stand Alone Tool Kit Common Macros

TRAP128

Source file: `maclib/TRAP128.mac`

Macro Format:

```
[label]    TRAP128
```

The `TRAP128` macro creates 128-bit disabled wait PSW's intended for placement at New PSW assigned storage locations. These PSW's allow the program to cease operation when an interruption class occurs for which no interrupt service routine is available.

Each interruption service routine trap PSW contains in its instruction address field the real location of the corresponding old PSW's assigned storage location.

Assembly Considerations:

- Each disabled wait PSW is assembled in the same order in which their respective assigned storage new PSW's occur in memory.

Execution Considerations:

- If not assembled at the new PSW assigned storage location, the program must move the PSW's to their respective locations before the "trap" is enabled.

Label Field Usage:

The label field is associated with the first assembled PSW.

Positional Parameters: None

Keyword Parameters: None

Stand Alone Tool Kit Common Macros

TRAPS

Source file: `maclib/TRAPS.mac`

Macro Format:

```
[label] TRAPS [ENABLE=[YES|NO|ONLY]]  
[ , PSW=[ [360|67|BC|EC|380|XA|E370|E390|label]] ]
```

The `TRAPS` macro generates and/or enables interrupt service routine trap PSW's.

Assembly Considerations:

- `TRAPS` requires a preceding `ARCHLVL` and `ARCHIND` macro in the assembly.
- If interrupt service routine traps PSW's are being enabled, a preceding base register must be established.

Execution Considerations: None

Label Field Usage:

The label field is associated with the first generated PSW (`ENABLE=NO`), or, otherwise, the first enabling instruction.

Positional Parameters: None

Keyword Parameters:

Keyword	Default	Description
ENABLE	YES	Controls what is generated for a specific instance of <code>TRAPS</code> : <ul style="list-style-type: none">• <code>YES</code> – causes PSW generation in-line and enabling instructions.• <code>ONLY</code> – causes only enabling instruction generation.• <code>NO</code> – causes only PSW generation.
PSW	None	<ul style="list-style-type: none">• When <code>ENABLE=ONLY</code>, this keyword specifies the location of the new PSW's in the assembly by referencing the label associated with a corresponding <code>TRAPS ENABLE=NO</code> macro and is required.• Otherwise, this keyword overrides the current assembly architecture level's PSW format.

Programming Note:

`TRAPS` is intended for run-time use only. `TRAPS` must not be used to directly assemble interrupt service routine PSW's into their assigned storage locations. Use `TRAP64` and/or `TRAP128` for that purpose.

`TRAPS ENABLE=YES` generates the PSW's in-line with the enabling instructions.

Stand Alone Tool Kit Common Macros

To separate the PSW creation from the enabling instructions in the assembly use:

```
TRAPS  ENABLE=ONLY, PSW=PSWS
```

other intervening assembly statements

```
PSWS    TRAPS  ENABLE=NO
```

Stand Alone Tool Kit Common Macros

ZARCH

Source file: `maclib/ZARCH.mac`

Macro Format:

```
[label]  ZARCH pair, cpur
          [, SUCCESS=label]
          [, FAIL=label]
```

The `ZARCH` macro changes the system architecture from ESA/390 to z/Architecture.

Assembly Considerations:

- `ZARCH` requires a preceding `ARCHLVL` macro in the assembly.

Execution Considerations:

- See the section “Side-Effects of Architecture Mode Changes” for execution considerations when using `ZARCH` macro.
- The condition code is set with the same conditions as described by the `SIGCPU` macro.

Label Field Usage:

The label field is associated with the first instruction generated by the macro when present.

Positional Parameters:

1. The `pair` positional parameter requires the identification of an even/odd general register pair used to the architecture change.
2. The `cpur` positional parameter requires the identification of another general register used by the architecture change.

Keyword Parameters:

Keyword	Default	Description
FAIL	none	The label to which control is passed upon successful change of the architecture. If omitted, control passes to the next sequential instruction.
SUCCESS	none	The label to which control is passed upon successful change of the architecture. If omitted, control passes to the next sequential instruction.

Programming Note:

If neither `FAIL` nor `SUCCESS` keyword parameters are supplied, control passes to the next sequential instruction regardless of the success or failure of the architecture change. The condition code can be tested by the program for success or failure and the reasons.

Stand Alone Tool Kit Common Macros

ZEROH

Source file: `maclib/ZEROH.mac`

Macro Format:

```
[label] ZEROH reg,bits
```

The `ZEROH` macro sets the high-order bits of either a 32-bit or 64-bit register to zero.

Assembly Considerations:

- `ZEROH` requires a preceding `ARCHLVL` macro in the assembly.

Execution Considerations: None

Label Field Usage:

The label field is associated with the first instruction of the macro when present.

Positional Parameters:

1. Identifies the general register whose low-order bits are being set to zero.
2. The number of high-order bits being set to zero.

Keyword Parameters: None

Programming Note:

`ZEROH` is useful when changing architecture mode and the contents of the high-order 33 bits must be zero to ensure a 31-bit address with 31-bit addressing mode does not result in an unexpected 64-bit address. To clear those high-order bits, use:

```
ZEROH reg,33
```

Stand Alone Tool Kit Common Macros

ZEROL

Source file: `maclib/ZEROL.mac`

Macro Format:

```
[label] ZEROL reg,bits
```

The `ZEROL` macro sets the low-order bits of either a 32-bit or 64-bit register to zero.

Assembly Considerations:

- `ZEROL` requires a preceding `ARCHLVL` macro in the assembly.

Execution Considerations: None

Label Field Usage:

The label field is associated with the first instruction of the macro when present.

Positional Parameters:

1. Identifies the general register whose low-order bits are being set to zero.
2. The number of low-order bits being set to zero.

Keyword Parameters: None

Stand Alone Tool Kit Common Macros

ZEROLH

Source file: `maclib/ZEROLH.mac`

Macro Format:

```
[label]  ZEROLH reg,bits
```

The `ZEROLH` macro sets the high-order bits starting at bit 32 of a 64-bit register to zero.

Assembly Considerations: None

Execution Considerations: None

Label Field Usage:

The label field is associated with the first instruction of the macro when present.

Positional Parameters:

1. Identifies the general register whose low-order bits are being set to zero.
2. The number of low-order bits being set to zero.

Keyword Parameters: None

Programming Note:

On a system having 32-bit registers, this macro is equivalent to `ZEROH`.

Stand Alone Tool Kit Common Macros

ZEROLL

Source file: `maclib/ZEROLL.mac`

Macro Format:

```
[label]  ZEROLL reg,bits
```

The `ZEROLL` macro sets the low-order bits starting at bit 32 of a 64-bit register to zero.

Assembly Considerations: None

Execution Considerations: None

Label Field Usage:

The label field is associated with the first instruction of the macro when present.

Positional Parameters:

1. Identifies the general register whose low-order bits are being set to zero.
2. The number of low-order bits being set to zero.

Keyword Parameters: None

Programming Note:

On a system having 32-bit registers, this macro is equivalent to `ZEROL`.

Stand Alone Tool Kit Common Macros

HOWTO be Architecture Sensitive

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.
- `ANTR` – 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.

Stand Alone Tool Kit Common Macros

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.

Stand Alone Tool Kit Common 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.

Stand Alone Tool Kit Common Macros

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.

Stand Alone Tool Kit Common Macros

HOWTO Define 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.
- `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 PSWFMT: PSWB PSWE PSWZ	ASAREA ASAZAREA	--	--

HOWTO Change Architecture Mode

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 `ASALOAD` macro. Alternatively, the program can, after control 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

Stand Alone Tool Kit Common Macros

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 and is no longer the sign bit. 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 architecture mode. The change in architecture mode causes the 64-bit register's contents to be interpreted as a positive integer rather than the negative integer 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 the instruction 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

Stand Alone Tool Kit Common Macros

Macro	Level	Register Bits	Starting Bit	ESA/390 or lower	Z/Architecture
	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 `-ts390x`, 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.

HOWTO Terminate 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	--

HOWTO Fly 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.

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

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

Stand Alone Tool Kit Common Macros

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.

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

Stand Alone Tool Kit Common Macros

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.