Table of Contents

Notices....5
Introduction....5
Overview....6
Compatibility Objectives....7
Getting Started....8
Step 1 – Installing Python....8
Step 2 – Testing the Environment....8
Invoking the Assembler....9
System Requirements....9
Environment Variables....9
Command Line Options....10
Assembler Controls....11
-t/--target ISA....11
--ccw FORMAT....11
--cp TRANS[=FILE]....12
--cpu MSLFILE=CPU....12
-e/--error LEVEL....12
-h/--help....13
--nest DEPTH....13
--psw FORMAT....13
Input Options....13
source....13
--case....13
-D SYMBOL[=VALUE]....13
Output Options....14
-d/--dump....14
-h/--help....14
-m/--mcall....14
-a/--addr SIZE....14
-g/--gldipl FILEPATH....14
-i/--image FILEPATH....15
-l/--listing FILEPATH....15
-o/--object FILEPATH....15
-r/--rc FILEPATH....15
-s/--store FILEPATH....15
-v/--vmc FILEPATH....15
--debug OPTION , --oper OPER, --pas N....15
ASMA Assembly Listing....15
Code Pages....16
Built-In Code Page Definitions....17
Local Code Page Modifications....18
codepage.py Command-Line Options....18
-a/--a2e....18

-c/--codepages....18
-e/--e2a....19
-p/--codepoints....19
--cpfile....19
--cptrans....19
--dumpa....19
--dumpe....19
--test....19
--write FILEPATH....19
Troubleshooting....19
Python Exceptions....20
Legacy Programs....21
Statement Format....21
D-type Constants....21
Binary and Hexadecimal Floating-Point Constants....22
Literals....22
Vector Instructions....22
Memory Image Output....23
Regions....23
Control Sections....24
Dummy Sections....24
Image, Region and Control Section Attributes....24
Positioning Binary Content When Using Hercules....25
Using the loadcore Command....25
Using the loadtext Command....25
Using the List-Directed ipl Command....26
Using the Image File....26
Using List Directed IPL....27
Using the Object Deck....28
Using the RC Script File....29
Using a STORE Command File....29
Assembly Language....31
Phase 0....31
Assembly Operation Identification....32
Phase 1....32
Phase 2....33
MSL Considerations....33
ASMA Input Statements....34
Names....34
D' Attribute....35
I' Attribute....35
K' Attribute....36
L' Attribute....36
M' Attribute....36

N' Attribute....36
O' Attribute....36
S' Attribute....37
T' Attribute....37
Labels....38
Symbolic Variables....39
Sequence Symbols....39
Self-Defining Terms....39
Binary....39
Character....39
Decimal....40
Hexadecimal....40
Quoted Strings....40
Literals....40
Case Sensitivity....41
Operand Values....41
Location Counters....42
Machine Instructions....43
Assembler Directives....43
AMODE....43
ATRACEOFF....44
ATRACEON....44
CCW....44
CCW0....45
CCW1....45
CNOP....46
COPY....46
CSECT....46
DC....47
DROP....49
DS....49
DSECT....49
EJECT....50
END....50
ENTRY....50
EQU....51
LTORG....51
MHELP....51
MNOTE....52
OPSYN....53
ORG....54
POP....54
PRINT....55
PSW....55

PSWS....56
PSW360....57
PSW67....58
PSWBC....59
PSWEC....59
PSW380, PSWXA, PSWE370, PSWE390....60
PSWZ....61
PUSH....63
REGION....63
RMODE....63
SPACE....64
START....64
TITLE....66
USING....66
XMODE....67
Macro Language....68
Macro Libraries....68
Macro Definition Mode....68
Macro Operation Identification....69
Macro Invocation....70
Macro Symbols....70
Subscripts....71
Variable Symbol Attributes....71
System Variable Symbols....72
Prototype Statement....73
Model Statements....73
Arithmetic Expressions....74
Logical Expressions....75
Macro Directives....76
ACTR....76
AGO....76
AIF....77
ANOP....78
GBLA....78
GBLB....79
GBLC....79
LCLA....80
LCLB....81
LCLC....81
MACRO....82
MEND....82
MEXIT....82
SETA....83
SETB....83

SETC....84
Appendix A - Instruction Source Formats....87
Syntax Summary by Instruction Format....87
Extended Mnemonics....91
Appendix B - Use of Machine Specification Language (MSL)....94
Instruction Formats....94
Statement Names....96
CPUX Objects....96

Notices

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

“Python” is a registered trademark of the Python® Software Foundation.

Introduction

Stand-alone mainframe bare metal programming needs are unique. They can be addressed by the GNU assembler, `as`, configured for s390 Executable and Linking Format (ELF) output, as is available with the Stand-alone Tool Kit (SATK).

GNU `as` itself has a number of annoying deficiencies for a mainframe assembler programmer familiar with proprietary assemblers.

The most notable are:

- no DSECT's,
- no USING and DROP statements,
- no cross reference listing,
- no auto alignment,
- no implied lengths,
- no backward ORG statement,
- extremely limited macro language, and
- no macro libraries.

Bare metal programming really needs to create memory content images or other output formats tailored to the bare metal environment (for example, emulator or virtual machine). GNU tools can be coerced into doing this, but not without external support and tooling. SATK provides this tooling but the GNU `as` limitations persist.

The z390 project has a much better assembler than GNU `as`, but for stand-alone

programming lacks privileged instructions. However, it only produces, in this context, an object deck. A loader is required to use this output. Such loaders are readily available. However, in some contexts, a memory image or other format is required and z390 is not capable of creating other output formats. External tooling would be required to do so.

Proprietary assemblers suffer from the same limitation as z390 with regards to output formats.

Overview

A Small Mainframe Assembler (ASMA) addresses the bare-metal programming needs for mainframes. It has the look and feel of a traditional mainframe assembler. It does not suffer from the limitations of GNU `as`. Familiarity with mainframe assembler technology and instructions is expected of the reader.

ASMA supports two related but **separate** languages:

- an assembly language for mainframe machine instructions, and,
- a macro language for dynamic generation of assembly language statements.

See the “Assembly Language” and “Macro Language” sections for specifics on the ASMA assembly and macro languages and where they diverge from traditional mainframe assemblers.

Within this document certain descriptions will be initiated by one of these expressions in **bold face** font. They have the implied significance described here:

- **ASMA Limitation**: describes a capability typically available with mainframe assemblers but unavailable in ASMA.
- **ASMA Specific Behavior**: describes a capability that diverges from typical mainframe assemblers or is unique to ASMA and unavailable in those other assemblers.

To comprehensively document such unique aspects of ASMA is a major goal of this document. However, it is possible that some differences are missed. If in doubt, and you do not see something discussed here, assume it is not supported. Alternatively, try it and see what happens.

Descriptions identified as “ASMA Limitations” are likely candidates for removal in future versions of ASMA, while no commitment is implied to do so.

> ***Before use of ASMA, the recommendation is strongly made that this document be reviewed allowing the user to assess what modifications to existing code or the user's own coding practices may be required.***

ASMA Limitation: ASMA does not have the concept of “open code”. Macro language directives are not recognized outside of macro definitions. Symbolic variables are only replaced when they occur within macro **model** statements. Conditional assembly is not supported outside of a macro. Wrap a legacy program that uses conditional assembly within a macro definition.

Compatibility Objectives

It is the objective of ASMA to provide an assembler that provides capabilities similar to legacy assemblers. In general a user of such legacy assemblers should find few “surprises” when using ASMA.

Nevertheless, no objective exists to provide identical capabilities of any legacy assembler of any vintage. Depending upon the coding practices the user has developed from use of such assemblers, adjustments may be required. Such adjustments should be well within the capabilities of an experienced assembler programmer, albeit for some, personally frustrating. The author apologizes ahead of time for such discomfort. The author believes that simple persistence in use will aid in adjusting.

If you intend to assemble a program developed with ASMA using a different assembler, avoid anything identified as **ASMA Specific Behavior**.

Programs developed with ASMA *should* assemble with legacy assemblers depending upon their vintage. But, implementation differences exist and some deviations may be encountered. Addressing such when found are considered enhancements and a failure to fully document **ASMA Specific Behavior**. No commitment is made here for incorporation of the enhancement and will be reviewed on a case by case basis.

Getting Started

This section assumes you have installed the SATK package on your system as described in the `SATK/README.txt` file.

Step 1 – Installing Python

Go to the **Python** site, https://www.python.org/, and follow its "Download" link for version 3.3 of **Python**, or later. ASMA is tested presently with **Python** version 3.4.

Step 2 – Testing the Environment

Review the information in the section "Command-Line Arguments" for how to invoke the `asma.py` script.

Change your current working directory to `SATK/tools`.

Enter:

```
[python] asma.py --help
```

If you have successfully established the environment, a listing of the script argument command-line options will be provided.

If not, you may be alerted to the wrong **Python** version or some other local environment response may occur.

Once you have successfully established the **Python** environment, ASMA and other **Python**-based tools can be used. Follow the section "Invoking the Assembler" in this manual for additional information on using ASMA.

Other manuals in the `SATK/doc` directory can be found for other available tools.

Invoking the Assembler

ASMA is invoked from a command line. The `--help` script argument summarizes the available arguments. Refer to the “Command Line Options” section for more information.

System Requirements

The assembler requires **Python** version 3.3 or later and will run on any platform on which the required level is installed. Install this version if it is not already available on the system intended to use ASMA. ASMA is available within and heavily dependent upon the tools provided by the Stand-alone Tool Kit, SATK. SATK is available from `github`:

https://github.com/s390guy/SATK

The **Python** directory search order is dynamically set by the tool by adding these directories:

```
${SATK_DIR}/tools/lang
${SATK_DIR}/tools/ipl
${SATK_DIR}/tools
```

`${SATK_DIR}` is the directory into which SATK is installed.

The `PYTHONPATH` environment variable is not normally required.

Environment Variables

Four search path environment variables consistent with the platform's conventions are used:

- `ASMPATH` – defines the search path for assembler `COPY` directives **and the input file**,
- `CDPGPATH` – defines the search path for a code page specification file,
- `MACLIB` – defines the search path for macro library defined macros (see the “Macro Libraries” section within the macro language description for details), and
- `MSLPATH` – defines the search path for MSL include statements. If not defined, the `MSLPATH` defaults to the directory `${SATK_DIR}/asma/msl`.

The `CDPGPATH` and `MSLPATH` environment variable is not normally required.

Each path environment variable uses the same approach to locate a file, searches are performed in this order:

1. Each of the directories in the order specified by the environment variable is searched.
2. If the file has not been located and the current working directory is not one of the specified directories in the environment variable, it is searched.
3. If the environment variable is not supplied, only the current working directory is searched.

By convention SATK ASMA source files use the `.asm` suffix, but are not required to do so.

One **Python** environment variable can effect performance: `PYTHONOPTIMIZE`. If set to a non-empty value, **Python** creates optimized **Python** byte code. This is equivalent to the **Python** command-line option `-O` and implied by the `-OO` option. Use of either this environment variable, or the command line option, may improve performance depending upon the size and content of the assembled source file. See the section "**Python** Exceptions" for more information.

Command Line Options

ASMA is initiated by a command line identifying the **Python** module `asma.py`. Refer to **Python** documentation for the mechanisms available on a specific platform for invoking a **Python** script and specifying the active `PYTHONPATH` environment variable if needed.

For `asma.py`, the `PYTHONPATH` environment variable should not be required. The `asma.py` script is designed to add any SATK **Python** containing directories required dynamically to the **Python** module search path.

Assembler command line options are specified as **Python** script file arguments. On some systems, the **Python** module itself can be specified directly on the command line:

```
asma.py [script file arguments]
```

Some systems may require first python itself be invoked on the command line:

```
python asma.py  [script file arguments]
```

or possibly

```
python3.3 asma.py [script file arguments]
```

Either form of command-line is supported by ASMA. On systems that support the first form, ASMA assumes that the path to the **Python** version 3.3 or later executable is:

```
/usr/bin/python3
```

Refer to **Python** platform specific documentation describing how **script file arguments** are provided on a command line. While the manner in which script file arguments are specified in a command line is platform dependent, the arguments themselves are platform independent. This section describes the script file arguments supported by `asma.py`.

For ASMA the script file arguments take the form of command line options with either a short or long form. Command line options with only a long form are not expected to be used under normal operation of the assembler. These options are used primarily for debugging purposes.

ASMA script file arguments fall into three broad categories:

- assembler controls,
- input options and
- output options.

A number of options specify a file. If a relative path or just the file name is specified, the file must be accessible from the current directory search path or an identified path environment variable.

Assembler Controls

Assembler controls influence the operation of the assembler.

-t/--target ISA

The `-t` or `--target` option specifies the instruction set architecture targeted by the assembly. The following MSL file and CPU are implied by the following values supported for `--target`. If omitted, the argument defaults to `24`. The implied MSL file must be located within either the default MSL directory or the specified directory search order in the `MSLPATH` environment variable. More information about MSL is provided in Appendix A.

The primary attributes of each value and implied MSL file and CPU are provided in the following table.

CPU Architecture	`--target`	MSL File	MSL File CPU	Implied `--addr`	XMODE PSW	XMODE CCW
System/360	s360	s360-insn.msl	s360	24	360	CCW0
System/370	s370	s370-insn.msl	s370	24	EC	CCW0
Hercules s/380	s380	s380-insn.msl	s380	31	380	CCW0
370-XA	370xa	s370XA-insn.msl	s370XA	31	XA	CCW1
ESA/370	e370	e370-insn.msl	e370	31	E370	CCW1
ESA/390	e390	e390-insn.msl	e390	31	E390	CCW1
ESA/390 on z/Architecture	s390	s390x-insn.msl	s390	31	E390	CCW1
z/Architecture	s390x	s390x-insn.msl	s390x	64	Z	CCW1
all (see Note 1)	24	all-insn.msl	24	24	EC	CCW0
all (see Note 1)	31	all-insn.msl	31	31	E390	CCW0
all (see Note 1)	64	all-insn.msl	64	64	E390	CCW0

Note 1. All supported instructions are recognized by the assembler for this target architecture. The selection of `XMODE PSW` and `CCW` values are those valid for the IPL PSW and IPL CCW's for the environment implied by the target argument. The `64` target value uses the syntax of the `IPTE` instruction in z/Architecture while the `24` and `31` target values use the original `IPTE` instruction syntax.

--ccw FORMAT

The `--ccw` option sets the initial execution mode CCW, overriding the expected CCW defined

for the CPU in the MSL database. Accepted values are:

`0`, `1`, and `none`.

A value of `none` removes the `XMODE CCW` setting, disabling the `CCW` directive.

--cp TRANS[=FILE]

Specifies the specific ASCII and EBCDIC character translation, `TRANS`, code page used by the assembler. If not specified, it defaults to the `94C` definition assumed to be available in the default code page file. Specify the optional `FILE` if a different code page file is used. See the "Code Page" section for details.

--cpu MSLFILE=CPU

The `--cpu` option identifies both the MSL file and the CPU within it for which the assembly is targeted. The two are specified separated without spaces by an intervening equal sign '`=`' as shown by this example:

```
--cpu s360-insn.msl=2020
```

-e/--error LEVEL

The `-e` or `--error` option specifies the level of error handling and method of reporting in use. Three values are supported:

- '`0`' – No error handling is provided. The assembler terminates immediately with a **Python** exception. Should only be used for diagnostic purposes.
- '`1`' – User errors are displayed to the user when detected by the assembler. Additionally, the error is provided in the listing with the statement in error and summarized at the end of the listing. Useful for problem isolation.
- '`2`' – User errors are provided in the listing with the statement in error and summarized at the end of the listing. Additionally the error summary is displayed to the user. This option is the default level of error handling and should be considered normal error handling.
- '`3`' – User errors are provided only in the listing with the statement in error and summarized at the end of the listing. Errors are not displayed to the user.

To the user, the apparent difference between options '`1`' and '`2`' is the sequence of displayed errors. When using option '`1`' errors may not be in assembly language statement sequence (because they are displayed when encountered). When using option '`2`', errors will be in assembly language statement sequence (because they are sorted after all errors have been identified). On occasion the number of errors may be too large for practical use of displayed errors. Option '`3`' limits errors to the listing file only. Redirection of displayed output to a file may be required in some cases

-h/--help

The `-h` or `--help` option displays the script file options available and exits.

--nest DEPTH

The `-n` or `--nest` option sets the maximum number of nested input sources. An input source may be a file, or a macro expansion. Defaults to 20.

--psw FORMAT

The `--psw` option sets the initial execution mode PSW format, overriding the expected PSW format defined for the CPU in the MSL database. Accepted values are:

`S`, `360`, `67`, `BC`, `EC`, `380`, `XA`, `E370`, `E390`, `Z`, and `none`.

A value of `none` removes the `XMODE PSW` setting, disabling the `PSW` directive.

Input Options

Input options identify for the assembler the source of input. Command-line source files and files included by the `COPY` assembler directive are stream text files, expected to utilize UTF-8 character encoding. **Python** universal newlines support will accept any of the common physical line termination character sequences. ASMA supplied text files utilize the standard Linux linefeed character to terminate source text files.

source

The source positional option identifies the input assembler source text file being assembled. It must be specified as the last script file argument and is required. Whatever is encountered as the last option is treated as the source text file with or without a path.

If a relative path or file name alone is used, the `ASMPATH` directory search order is used to locate the file as with a `COPY` directive.

--case

Enables case-sensitive handling of assembler language labels, macro language symbolic variables and macro language sequence symbols. By default input is case-insensitive.

-D SYMBOL[=VALUE]

The `-D` option establishes a global character symbolic variable and, if provided, assigns it a character string value. Supplying only a symbol is equivalent to a `GBLC` macro language directive. Providing a value is equivalent to the `GBLC` being followed by a `SETC` macro language directive.

This option performs a role similar to the system symbolic variable `&SYSPARM` but allows any

symbol other than those already utilized by ASMA as system symbolic variables to be specified. Unlike the `&SYSPARM` variable, symbols defined by the `-D` option are normal read-write symbols and may be altered by a defined macro. As with all user defined global symbolic variables, a macro must declare its usage before accessing the symbol.

Output Options

Output options influence the output created by the assembler.

-d/--dump

The `-d` or `--dump` option cause the assembly listing, if itself is not suppressed, to include the image file to be included in the format of a storage dump. Both hexadecimal object and ASCII and EBCDIC characters are interpreted in the dump. Memory addresses bound to each region and positions of the image content relative to the start of the binary image are both provided in the dump.

-h/--help

The `-h` or `--help` option displays the script file options available and exits.

-m/--mcall

The `-m` or `--mcall` option causes inner macro statements to be included within the listing when statement printing is currently enabled by the `PRINT ON` directive. Normally inner macro statements encountered during a macro's invocation are suppressed from the listing.

-a/--addr SIZE

The `-a` or `--addr` option overrides in the listing, the address size specified within the MSL for the target CPU. Only four values are accepted: 16, 24, 31, or 64. The option applies to the statement area of the listing and the optional image file dump.

ASMA Specific Behavior: This option influences the size of address fields used by the assembler listing. If you prefer to standardize on a single address field size, use the command line argument to override values resulting from the target CPU's definition.

-g/--gldipl FILEPATH

The `-g` or `--gldipl` option specifies a the file to which a generic list directed IPL file is written. Files participating in the list directed IPL are written to the same directory. If not specified, all list directed IPL files are suppressed. See the “Using List Directed IPL” section for details.

-i/--image FILEPATH

The `-i` or `--image` option specifies the file to which the binary image is written. If not specified, the image file is suppressed. See the “Using the Image File” section for details.

-l/--listing FILEPATH

The `-l` (lower-case L) or `--listing` option specifies the file to which the assembly listing is written. If not specified, the assembly listing is suppressed. Familiarity with mainframe assembler listings is expected to make the structure and content of the listing self-explanatory.

-o/--object FILEPATH

The `-o` pr `--object` option identifies the file to which a loadable object deck is written. If the option is not specified, an object deck is not created. See the “Using the Object Deck” section for details.

-r/--rc FILEPATH

The `-r` or `--rc` option identifies the file to which a Hercules RC script file containing real storage alter commands is written. If the option is not specified, the RC script file is not created. See the “Using the RC Script File” section for details.

-s/--store FILEPATH

The `-s` or `--store` option identifies the file to which a series of real storage STORE commands is written. If the option is not specified, the command file is not created. See the “Using a STORE Command File” section for details.

-v/--vmc FILEPATH

The `-v` or `--vmc` option identifies the file to which a series of real storage STORE commands is written. If the option is not specified, the command file is not created. See the “Using a STORE Command File” section for details.

--debug OPTION , --oper OPER, --pas N

These three options control various forms of debug output. Under normal circumstances, they should not be used. The `--help` option lists available values. Debug output may be limited or non-existent if the **Python** environment variable `PYTHONOPTIMIZE` or **Python** command line options `-O` or `-OO` are used.

ASMA Assembly Listing

ASMA assembly listings are created as ASCII character stream files. The ASCII form-feed

character initiates a new page. Individual detail lines are terminated with the platform specific line-termination character(s) as supplied by **Python**'s universal newlines facility:

https://docs.python.org/3/library/functions.html#open

ASMA uses **Python** universal newlines for all text files created by the assembler. Each page consists of 55 lines, including the title line.

ASMA assembly listings contain as many as seven distinct reports depending upon the assembly and command line options. The entire listing is controlled by the `--listing` command-line script argument. If omitted, no listing is produced.

The following table identifies the individual reports and the controls that influence the generation of the report within the listing. Reports for which no control is provided are always produced if the listing itself is generated.

Report	Control	Description
1	none	Assembled statements and their generated binary content
2	none	Symbol table information and symbol cross-reference
3	none	Macro cross-reference listing
4	none	Image map reporting image and memory residency of the image, each region within the image and each control section within each region.
5	`--dump`	Image content in the format of a memory dump if `--image` also specified.
6	`--dump`	Generated deck records if option `--object` also specified.
7	none	List of referenced files by number. File numbers are reported in the error report.
8	none	Error report. May contain only informational messages or no errors when none encountered.

Other forms of output are all text files and are not added to the listing. Direct inspection of the generated files is required to see the content.

Code Pages

ASMA input is **ASCII** based. Handling of input text is controlled by the **Python** support of Unicode. All input text files are treated by **Python** as UTF-8 character encoding as implemented by **Python**. Support for a different input encoding, while supported by **Python**, is not presently available to an ASMA user.

ASMA Limitation: The legacy practice of enclosing arbitrary values within a character self-defining term is inhibited by **Python**'s own text handling. Such character sequences must be replaced by either a decimal, hexadecimal or binary self-defining term.

When assembling character constants or referencing character symbolic variables in macro directives, the assembler must know how to convert ASCII characters presented to it into EBCDIC characters. Conversely when interpreting binary data as characters, the assembler must know the character represented by the binary data. The `codepage.py` module provides this support.

`codepage.py` provides both the internal support for code page usage and a command line interface for displaying useful information about code page definitions. Code page definitions follow the same general coding rules as are used with MSL files. A set of built-in definitions constitute the default character sets and translation options. Each translation definition utilizes an ASCII and EBCDIC code page definition to construct three translation tables:

1. ASCII-to-EBCDIC translation,
2. EBCDIC-to-ASCII translation, and
3. Binary interpretation translation.

ASCII/EBCDIC translation tables convert between the two code sets based upon those characters **shared** between them. Unshared or undefined code assignments are not translated.

Binary interpretation translates all single byte binary values to an ASCII character. Four approaches may be used for these conversions when an assignment exists for a code point:

1. Interpret only ASCII codes from the ASCII code page,
2. Interpret only EBCDIC codes from the EBCDIC code page,
3. Interpret both ASCII and EBCDIC codes from their respective code pages using the ASCII code assignment when a code assignment exists in both code pages, or
4. Interpret both ASCII and EBCDIC codes from their respective code pages using the EBCDIC code assignment when a code assignment exists in both code pages.

Unassigned values utilize a default “fill” character specified within the definition. The default definitions use an ASCII period, '`.`', as the fill character.

Built-In Code Page Definitions

The built-in definitions identify a single set of characters. These are the characters specified by an architecture reference summary manual as the '94C' assignments in its “Code Assignments” section with the exception of four characters:

- E0, the Eight Ones character;
- NSP, the Numeric Space character;
- RSP, the Required Space character; and
- SHY, the Soft Hyphen character.

The built-in definitions provide the four translation options using the identification:

- `94C` – Interpret EBCDIC only (approach 2 above),
- `94Ca` – Interpret ASCII only (approach 1 above),
- `94Cea` – Interpret ASCII and EBCDIC with EBCDIC preferred over ASCII (approach 4 above), and
- `94Cae` – Interpret ASCII and EBCDIC with ASCII preferred over EBCDIC (approach 3 above).

Any of these translation definitions may be used in the `--cptrans` command line option. `94C` is the default.

It is possible when using the set of 94C definitions to generate unrecognized ASCII characters on a typical PC keyboard. These will appear in assembled object code and macro language character symbolic variables as unchanged.

Local Code Page Modifications

The `codepage.py` module was designed specifically with local modifications in mind. The built-in definitions can be output to a file using the `codepage.py --write` command line option. Once output, the file may be locally modified to satisfy local code page needs. The local file and translation code page is then referenced in the ASMA `--cp` command line option for use by the assembler.

Two character set groups are provided within the default definitions for local changes: EBCDIC-local and ASCII-local. They appear at the beginning of the file. Apply your local definitions there. Because duplicate definitions are not allowed, in some cases a supplied character definition may need to be turned into a comment.

The `codepage.py` command line options, described in the next section, are provided to assist in verifying the expected results.

Changing the supplied default definitions from within the `codepage.py` module is also possible but not recommended. The default definitions are embedded within the `codepage.py` **Python** module as the global variable `default` defined near lines 45-302.

codepage.py Command-Line Options

-a/--a2e

If present, print the ASCII-to-EBCDIC translation table of the selected translation ID as a 16-by-16 matrix.

-c/--codepages

If present, print the ASCII and EBCDIC code pages of the selected translation ID as a list sorted by character name.

-e/--e2a

If present, print the EBCDIC-to-ASCII translation table of the selected translation ID as a 16-by-16 matrix.

-p/--codepoints

If present, print the list of ASCII and EBCDIC characters, by name, assigned to each code point.

--cpfile

Specifies the code page file defining the ASCII and EBCDIC character translations available to the command line utility. The `CDPGPATH` environment variable defines the directory search path used to locate the file. If not specified the built-in translation definitions are used.

--cptrans

Specifies the specific ASCII and EBCDIC character translation definition for which the output options apply. If omitted the id `94C` is used.

--dumpa

If present, print the binary interpretation table of the selected translation ID as 16-by-16 matrix. Translation is to ASCII output characters.

--dumpe

If present, print the binary interpretation table of the selected translation ID as 16-by-16 matrix. Translation is to EBCDIC output characters.

--test

Perform a simple translation test on the value supplied with the argument.

--write FILEPATH

Output to the default code page definitions to the specified file.

Troubleshooting

When unexpected problems occur take the following steps:

Do not set the **Python** `PYTHONOPTIMIZE` environment variable. This option causes the removal of internal value checking and suppression of some debug or tracing results. Creation of optimized byte-code removes assert statements and others operating under control of the **Python** variable `__debug__`. Normally these are not needed and assembly

times are improved.

Modify the `--error` reporting level to improve error reporting. The default, `2`, causes errors to be reported at the end of the assembly and in the listing. Unanticipated **Python** exceptions will cause the assembler to terminate before a listing is produced and errors are reported. Use an error level of `1` to have errors reported when they are detected. Use a level of `0` to disable **Python** exception handling. This may result in the initial detected error being displayed.

If the failing statement can be identified, use the `--oper` to trace processing of a specific operation. More general processing can be enabled by using `--debug`. Both of these options can generate excessive output. Output from these options are not documented and is only meaningful if the related **Python** code is understood. If these two options are used, it is recommended to create a test file that only contains the problem statement or statements to reduce the output.

If you remove the definition for the `PYTHONOPTIMIZE` environment variable but debugging or tracing information is not appearing, ensure any directories named `__pycache__` are removed. This directory is where **Python** stores compiled byte-code. If the cached byte-code has removed optional error checking code, the directory's removal will force recreation of the **Python** byte-code with the error checking code present.

Python Exceptions

Generally any **Python** exception raised by the assembler is a bug.

The exception to this statement is when the command line option `--error 0` is in effect. This option disables all internal trapping of **Python** exceptions. When this option is in use, `AssemblerError` exceptions should be considered as normal. These exceptions are used when a user error is encountered. All other exceptions are bugs.

There are occasions where the reason a statement is reported in error is not entirely clear. This can result from the trapping of other exceptions that become translated into an `AssemblerError` exception. By using command line `--error 0` option the triggering exception may be reported. In this case, the exact location of the inner error will be reported facilitating resolution.

Regardless of the `--error` option in effect, a `ValueError` or `AssertionError` exception is a bug. Internal checks that result in unanticipated conditions raise a `ValueError` or `AssertionError` exception. They are never caught and always immediately terminate the assembler. **Python** itself may raise a `ValueError` or other exceptions. Uncaught exceptions raised by either **Python** or the assembler always terminate the assembler and always require correction.

Legacy Programs

This section brings together in a single place some of the areas of adjustment that are required by ASMA to an existing assembly source program. Refer to details within the document for more information.

Most programs accepted by a proprietary assembler will likely require some modification for ASMA.

The following sections are based upon limited experience in using ASMA with legacy programs, but the areas discussed either reoccur or deserve explicit mention.

Although obvious, any assembler directives or constant types not supported by ASMA must be addressed.

Statement Format

ASMA expects variable length input lines. Generally the presence of sequence columns become interpreted as comments. **Python** supports a universal newline strategy which makes ASMA insensitive to the underlying platform's standard line termination sequence.

Comment statements start with an asterisk, `*`. If the asterisk is preceded by spaces or other “white space” characters such as tabs, the statement will be found in error.

ASMA expects continuation lines to be indicated by a backslash, `\`, in the last character position of the variable-length line.

Only in the case where a legacy program utilizes both sequence columns and has a continuation in column 72 will ASMA not recognize the continuation. Plans exist to support 80-column input records in the legacy format but is not yet available. For such statements the sequence columns must be removed and the continuation character in column 72 must be changed to a backslash character.

If problems result with continuation lines, remove continuation by creating a single long line.

The alternate continuation conventions available with legacy macros is supported by ASMA but has had limited testing.

D-type Constants

A common legacy practice is to utilize a `D`-type constant for doubleword alignment and placement of binary zeros into an 8-byte field. ASMA treats a `D`-type constant as a synonym for the `FD`-type constant. Floating point nominal value syntax is not supported. Only integer nominal values are recognized. When hexadecimal floating-point constants are supported, `D`-type constants will be supported as long hexadecimal floating point constants.

Binary and Hexadecimal Floating-Point Constants

ASMA lacks support for binary and floating point constant types. Although floating point instructions are supported, the creation of binary or hexadecimal floating point constants for them is not. The only available solution is for such constants to be replaced by `X`-type constants. An assembly listing from another source could provide the required content. Resolution of this limitation is an eventual goal.

Decimal Floating Point constants are fully supported.

Literals

While literals are supported, only basic support within machine instructions is available. Where usage of a literal is not supported it must be replaced with a constant defined by a `DC` directive with a label. The label is required to reference the constant in these situations.

Vector Instructions

z/Architecture vector instructions are fully supported. Legacy (pre-z/Architecture) vector instructions are not presently supported by ASMA. If the reader has a need for legacy vector instructions, the reader is encouraged to work with the author in developing their support within ASMA. MSL is the path to such support.

Memory Image Output

ASMA Specific Behavior: Everything in this section is unique to ASMA or diverges from legacy practices.

The output of the ASMA assembler is not the typical object deck composed of ESD, TXT, etc. records. Internally, ASMA creates an “image” of binary content. Each of the different output formats exports the internal representation to one or more files in different ways.

The internal binary content consists of:

- One or more regions of binary content, each with an assembled starting memory location, and
- Each region consists of one or more control sections.

The order of the regions within the image is the order in which the `START` directive initiating the region is encountered. Correspondingly, the order of the control sections composing a region are the order in which the control section's initiating `CSECT` directive is encountered. Control sections are aligned to the next eight-byte address boundary when added to the region. Address constants are ultimately bound to the region's starting address dictated by the control sections location within the region. Because addresses are bound to absolute locations, no constraints exist on addressing a location in one control section or region from another.

Conceptually the image is something one might burn into a ROM on more modern systems. It is one large “chunk” of binary data. From an external point of view the image file has no structure. The program in the image can impose its own structure on the image content through the use of the assembly directives `START`, `REGION` and `CSECT`.

To illustrate, an image could contain an installer, when executed, installs another program (also part of the image) on a device that itself could be used as the IPL source for the installed program. Some object formats, for example, an Executable and Linking Format (ELF) executable file, require external tooling to accomplish the same task. Such tooling is available within the SATK through use of the `iplmed.py` utility. Such tooling limits the creation of IPL media to the ELF's abilities and standards. The image file approach offers much greater flexibility in this area. The image structuring directives allow the program to recognize and specify the residency requirements of different portions of the image.

Regions

A region contains binary content bound to a starting memory address identified in the `START` directive that initiates the region's creation. Unlike typical mainframe assemblers, multiple `START` directives are possible allowing the creation of multiple “regions”. Regions are placed physically contiguous within the image in the order in which the `START` directives that initiate them are encountered. A named region is identified by the `region` operand of its associated `START` directive. An unnamed region is created when the associated `START` directive has the

region operand omitted. A new START directive causes the following content to be placed within the newly initiated region and newly created control section. Content within a previously created region can be continued by use of a REGION directive.

Control Sections

Each region is populated by one or more control sections each of which is identified and initiated by a CSECT or START directive. A new control section is placed within the currently active region under construction with double word alignment. Like regions, a new control section will cause all succeeding content to be placed in the new control section. Content can again be placed in the previous control section by coding a CSECT directive with the label field containing the name of the control section being continued or omitted when the unnamed control section is targeted. Portions of a control section are made contiguous within the control section even if the assembler statements creating the content are not contiguous.

Control sections can not be created outside of a region. Once initiated within a region, all of the control section's content will be placed within the region. By continuing a control section, automatically the control section's region becomes active. Correspondingly, when a region is continued by a REGION directive, the previously interrupted control section becomes the control section into which new content is placed. A succeeding CSECT directive will change the active control section to another and to its associated region.

Dummy Sections

Dummy sections are not associated with any region and are never explicitly bound to a memory address. Only section relative addresses are ever assigned to symbols within a dummy section.

Image, Region and Control Section Attributes

The image, each named region and each named control section are identified by a symbol. Each symbol has certain attributes accessible from within the program. The symbol associated with a control section or region is specified in the label field of the respective initiating CSECT or START directive. The image is automatically assigned the name IMAGE in all upper case. By its early definition and the restriction on duplicate symbols, this name is really a reserved symbol. The only such symbol in the system.

All symbols have associated with them a value which may be an address or an integer value. In the case of the symbols used for an image, region or control section, the value is the starting absolute address as bound at the completion of Pass 1. The address assigned to the image is the address assigned in the first START directive of the program. This address is effectively the address to which the image expects to be placed in memory. The length attribute, referenced by preceding a symbol with the sequence L', is the total length in bytes of the image, region or control section.

Unlike other symbols, each image, region and control section symbol has an additional

attribute, the “image” attribute. Similar to the length attribute, it is referenced by preceding the symbol with the sequence `I'`. The image attribute provides the displacement from the start of the image of the corresponding symbol. Through this attribute a program can locate a region or control section within the image independent of its bound memory address.

The unnamed region's or unnamed control section's attributes are not accessible to the program because a symbol is not associated with either. Only a named region's or named control section's attributes are accessible to the program via the assigned symbol.

Positioning Binary Content When Using Hercules

How are you going to load the memory content into memory? The answer to that question drives which form of output you should select. This section answers the question from the perspective of using the Hercules mainframe emulator.

Some of this information is repeated with more details in the other related sections. This section tries to give simple guidance for getting what you want out of ASMA.

Using the `loadcore` Command

The Hercules `loadcore` command uses a single file. ASMA `--image` is the ideal output format for creating the binary content. However, only one `START` directive should be used creating only one region. When only one `CSECT` directive is used, binary content can be positioned anywhere within the image file by means of `ORG`. Binary content can overlay other content within a control section but not between control sections.

Regions within the `--image` output file never overlay another region and **do not get placed within the file based upon the region's assembled addresses.** Use of multiple regions with `--image` is not recommended unless you really understand the implications. When the image file's size is different than you expect or the content is in locations you do not expect, this is likely the case.

The `POS` column in the Image Map of the assembly listing identifies where each region and control section is placed in the image. The `ADDR` column indicates the assembled starting address of the regions and control sections. The `POS` column controls where in memory the region or control section is placed by the `loadcore` command, not the `ADDR` column. Use `--dump` option to see the image file content as written to the file included in the listing.

Using the `loadtext` Command

The `--object` output format is intended for use with the Hercules `loadtext` command. All TXT records created by the `--object` output format are based upon the assembled addresses of the statements. As such, when loaded, binary content from any region can overlay content from any other region as determined by the order in which the statements are assembled.

The `ADDR` column in the listing Image Map indicates where into memory each region and control section is placed. The `POS` column has no influence over how the TXT record content is loaded. Use `--dump` option to see the TXT records created by the `--object` output format in the assembly listing.

Using the List-Directed `ipl` Command

The `--gldipl` output format is designed for use with the Hercules list-directed `ipl` command. Each region is placed in its own list-directed file with its own entry in the list control file. The regions are loaded in the sequence in which they are created at their assembled starting address.

The `ADDR` column in the Image Map indicates where into memory each region and control section is placed. The `POS` column has no influence over how the TXT record content is loaded.

Using the Image File

The image file output format places each region physically adjacent to its next region regardless of the region's assembled address, in order, without consideration of the region's starting address. Making use of the image file requires the program, presumably in the first region to understand this and locate other regions accordingly. Areas not initialized by the assembly, for example by `DS` directives or forced by alignment are filled with binary zeros.

The image file contains no structure discernible externally. It is best suited to either a single region program or a program that understands how to locate and utilize the other regions. For multiple region programs, the list-directed IPL output option is recommended.

The image file is anticipated to be loaded into memory at the address specified in the first `START` directive. If the expectation is that the image file will be placed into memory and that a restart interruption will be used to enter it for execution, then the image file must be constructed with this in mind. If the image file is intended to contain the Restart New PSW, the image file must expect to be loaded at the memory address of the Restart New PSW. For systems not in z/Architecture ® mode, the assigned storage location of the Restart New PSW is address 0. For systems in z/Architecture mode, the assigned storage location is address 288, or in hexadecimal, 0x120. This drives the operand of the `START` directive to be either `X'0'` or `X'120'` (or if decimal is preferred, `0` or `288`). At the location of the Restart New PSW within the image file, must be a properly formatted PSW that will initiate running of the program within the image.

An image file constructed this way could also utilize list directed IPL. However, because the system will not be in z/Architecture mode when the IPL PSW takes control, the starting location of the first region must be 0 at which location must be a PSW generated by the assembly. Supplying the content of the list used during the IPL process is the responsibility of the user of the image file. See the section “Using List Directed IPL” for more information.

Once loaded into storage, it becomes the responsibility of the loaded program to properly

locate within storage any regions other than the first contained in the image. The region or control section symbol `M'` and `L'` attributes are intended to assist the loaded program in performing this content relocation.

Hercules `loadcore` command can load an image file into emulated main storage. Additional commands related specifically to the loaded image file are required to start its execution. The next section, “Using List Directed IPL”, provides a more error free method that will directly execute the loaded image.

Using List Directed IPL

The list directed IPL option outputs each region separately, and, via the control file maintains the starting address of the region when loading the region. Areas not initialized by the assembly, for example by `DS` directives or forced by alignment are filled with binary zeros. It is ideally suited for multiple region programs.

List directed IPL allows one of more files to be loaded into storage with control passed to the loaded content by causing the IPL PSW at address 0 to become the active PSW. The content of the directory is oriented towards use by the Hercules `ipl` console command with a file path. Refer to Hercules documentation for details. Multiple platforms may support list directed IPL. It is the user's responsibility to position the list directed IPL files in a compatible location and identification when using a platform other than Hercules.

The `--gldipl` option specifies the file into which IPL list is written. This option writes a file to the same directory as the IPL list file for each **region** defined in the assembly. Each region file uses the region's name as specified from its `START` directive and file extension of `.bin`. The unnamed region may not be used with list directed IPL. The IPL list file has the name of the image with the file extension provided in the command line option. It is the IPL list file that is specified in the Hercules `ipl` command.

Because the IPL function automatically uses the IPL PSW at absolute storage location 0, an assembly intended for use with list directed IPL must ensure a PSW is placed in storage at address 0 by the loaded regions. The PSW may be the only content of the region starting at address 0 or part of additional region content.

To illustrate this structure, assume an assembly contains two regions. The `IPLPSW` region starts at address 0 and only contains a 64-bit PSW. The main program region is `PROGRAM` and starts at location `X'2000'`.

The following example uses a UNIX-like environment, but ASMA runs on any platform supported by **Python** with **python** tailoring file names to the local environment. The user `johndoe` uses the following command line option in the assembly:

```
--gldipl test/program/test.ipl
```

All of the list directed IPL files will be written to the `/home/johndoe/test/program` directory. A relative directory is expanded to an absolute path. Three files will be placed in this directory:

`test.ipl` – the IPL list used during the IPL function

`IPLPSW.bin` – the 8-byte PSW

`PROGRAM.bin` – the actual program

The IPL list file (`test.ipl`) will contain two lines of text, as follows:

```
PROGRAM 0x2000
IPLPSW 0x0
```

The sequence of regions in the IPL list file, and hence the sequence in which regions are loaded into memory, is dictated by the sequence in which the regions are defined in the assembly.

The actual IPL command used with Hercules, likely placed in a RC script file or the Hercules configuration file, would be:

```
ipl test/program/test.ipl
```

The Hercules list directed IPL command does not support absolute paths on some environments. This statement assumes the current working directory is `/home/johndoe`.

Using the Object Deck

The object deck option creates object TXT records based upon the binary content of the assembled statements and their binding locations. The object deck is suitable for use with one or more regions. Areas of uninitialized content remain unchanged.

An object deck is created only when the `--object` command line option is specified. The deck contains one or more TXT records and an END record. TXT and END records are limited to three-byte address fields. If the assembly contains object code at any location with an address larger than `X'FFFFFF'`, the object deck is suppressed with a warning message. The END record contains the entry point identified by the last `ENTRY` directive. If no `ENTRY` directive is present in the assembly, the object deck is suppressed.

TXT records will only contain object code generated by assembly machine instructions or directives. Other than the address of the TXT record content, regions and control sections have no other influence on the output. Areas not explicitly initialized are not contained in the TXT record binary content. This means that areas defined by `DS` directives or gaps created due to implied alignments are not filled with binary zeros as is the case with the binary image file. Because control is passed to the loaded content by an object deck loader, an explicit PSW in storage is not required for execution entry to the TXT content. A separate card-based loader that may be the target of an Initial Program Load function is required and, if used, must precede the actual object deck card images created by this option.

The Hercules `loadtext` console command may also be used to load the content into storage. Unlike the card-based loader, a Restart New PSW would be required to manually pass control to the loaded program. A manual, or script driven, restart interruption would be required to actually cause control to be given to the program.

The object deck created by the `--object` option is not suitable for use with a linkage editor. It is only usable with a loader facility supporting TXT and END records.

Using the RC Script File

The Hercules emulator allows storage to be loaded using a real storage alter command, the Hercules `r` console command. The `--rc` option creates a script file containing a series of real storage alter commands that have the effect of loading the assembly content. As with the object deck, only areas explicitly initialized are contained in the real storage alter commands. Areas defined by `DS` directives or gaps created due to implied alignments do not cause real storage to be altered. Other than supplying the address of the storage alter command, regions and control sections have no other influence on the output. Each command takes the form of:

```
r address=hexdata
```

Because Hercules allows a script file to include other script files, the file created by the `--rc` option may be directly included in another file.

Depending upon how the user of the script file expects to pass control to the loaded storage content, a Restart New PSW may be required at real storage address 0 (`X'0'`) or 288 (`X'120'`). When using a script file, the easiest way to achieve this is by creating a region with the starting address of 0 or 288 into which is assembled the Restart New PSW. Other mechanisms exist for starting the CPU executing with Hercules console commands. An assembly generated Restart New PSW is just one way to achieve this without more error prone mechanisms.

Using a STORE Command File

A STORE command file contains a series of STORE commands that alter real storage content using a mainframe virtual machine environment. As with the object deck, only areas explicitly initialized are contained in the real storage alter commands. Areas defined by `DS` directives or gaps created due to implied alignments do not cause real storage to be altered. Other than supplying the address of the storage alter command, regions and control sections have no other influence on the output.

Two options create a STORE command file: `--store` and `-vmc`. The only difference between the two files is that the output from the `--vmc` option expects to be executed from a mainframe virtual machine environment. Each command within the `--vmc` generated STORE command file starts with `CP`. Output from the `--store` option does not contain the `CP` sequence. Each command takes the form of:

`STORE R S address hexdata` (for output from the `--store` option), or

`CP STORE R S address hexdata` (for output from the `--vmc` option).

Depending upon how the user of the STORE command file expects to pass control to the loaded storage content, a Restart New PSW may be required at real storage address 0

(`X'0'`) or 288 (`X'120'`). When using a STORE command file, the easiest way to achieve this is by creating a region with the starting address of 0 or 288 into which is assembled the Restart New PSW. Other mechanisms may exist for starting the CPU within a virtual machine environment. An assembly generated Restart New PSW is just one way to achieve this without more error prone mechanisms.

It is the user's responsibility to move the contents of the STORE command file into the virtual machine environment for actual execution of the real storage altering commands.

Assembly Language

ASMA utilizes three phases in two passes during its processing. An error can cause immediate failure or all errors can be reported at the end of the assembly depending upon the command line options selected.

Phase 0 and Phase 1 constitute the first pass by the assembler of the input statements. Statistics for phases 0 and 1 are identified as “Pass 1.” Phase 2 constitutes the second and final pass by the assembler of its statements. Statistics for phase 2 are identified as “Pass 2.”

Following Pass 2, output processing occurs. This includes generation of the assembly listing and any other output formats requested.

Phase 0 is performed on each statement. When phase 0 is successful, processing then proceeds to phase 1 for the same statement. Upon successful completion of phase 1, the next input statement is processed for phase 0, etc.

Internally, the ASMA code continues to refer to “Pass0”, “Pass1”, or “Pass2”. Rather than changing all such references in comments and method names, “Pass0” actually refers to “Phase 0” rather than a true pass of the statements.

Phase 0

All text statements enter the assembler in Phase 0.

Empty lines and comment lines are recognized with no further processing.

Statements are broken up into fields: label, operation, and operands with comments.

Operations are identified. See the “Operation Identification” section for details.

Statements containing an operation field for the following directives are processed in this phase: `ATRACEOFF`, `ATRACEON`, `COPY`, `MACRO`, `MHELP`, `OPSYN`, `PRINT`, `SPACE`, `TITLE` and `XMODE`.

Macro definitions are created and macro expansions are invoked. See the “Macro Language” section for details.

Other statements are analyzed as follows:

1. Symbolic variable substitution is performed. See the “Model Statements” section for details.
2. Assembler directives and machine instructions are recognized and execution mode options applied.
3. Parsing of operands into expressions as required by the identified operation. Expressions are not evaluated at this time, just parsed. Calculating the value of expressions occurs in either Phase 1 or Phase 2 depending upon the directive.

4. Comments on a statement are ignored.

Source statements brought into the input stream by the `COPY` directive occurs in Phase 0.

Assembly Operation Identification

ASMA statement processing occurs in one of either two modes, driven by the operation identification processing within the current processing mode:

- assembly mode or
- macro definition mode. See the “Macro Operation Identification” section for details.

In assembly mode, operation identification occurs in these steps:

Step 1 – Apply `OPSYN` operation synonym

A currently defined operation synonym is selected. If the operation has been deleted, processing proceeds directly to Step 5.

Step 2 – Identify the machine instruction. If unsuccessful proceed to Step 3.

Step 3 – Apply `XMODE` definitions to operation, if applicable

Step 4 – Identify the assembler directive. If unsuccessful proceed to Step 5.

Step 5 – Identify a macro definition. If unsuccessful proceed to Step 6

Step 6 – Access the macro library path and attempt definition of the macro.

If all of these steps fail, the operation identification results in an error condition.

Phase 1

For instructions and other content producing directives, their bound locations are established and the size of their content is determined, although the content itself is not.

For labels, they are assigned a bound address of the current location of the current active control section.

For assembler directives not producing image content, they are acted upon at this point:

`CSECT`, `DS`, `DSECT`, `EJECT`, `EQU`, `ORG`, `REGION`, `START`.

Calculations performed by any of these directives require referenced symbols to already be defined, namely `EQU`, `ORG` and `START`.

At the end of Phase 1, the size of each statement's content and the size of each control section has been established. All statements without errors are processed by the next phase.

Phase 2

Each control section without an explicit starting location is established based upon its alignment attribute, its sequence within the assembly and its preceding control section's ending address.

Each symbol is bound to its final address based upon its position within its control section. The section “linking” process is now complete.

Content is created by evaluating all remaining expressions in the directives and processing the instruction. Content creating and content influencing statements of the assembler are completed during this pass:

`CCW`, `CCW0`, `CCW1`, `DC`, `END`, `DROP`, `PSWS`, `PSW360`, `PSW67`, `PSWBC`, `PSWEC`, `PSWXA`, `PSWE370`, `PSWE390`, `PSWZ`, `USING`, and all machine instructions.

At the completion of Phase 2, all requested output formats are created. Each statement's content is placed in its location within its control section at its location. See the various sections on using each of the various output formats.

ASMA Limitation: ASMA does not have the concept of “open code”. Macro directives are not recognized outside of macro definitions. Symbolic variables are only replaced when they occur within macro model statements. Conditional assembly is not supported outside of a macro. Wrap a legacy program that uses conditional assembly within a macro definition.

ASMA Limitation: Lookahead mode is not supported. Conditional assembly statements `AIF` or `AGO` are not supported in “open code.” “Open code” must be wrapped within a macro definition for these statements to be effective.

ASMA Limitation: Expressions in assembler statements are limited to arithmetic computations: addition, subtraction, multiplication and division. Logical operators, for example, “and”, “or” and comparisons, while supported by the macro language, are not supported by the assembler. These are separate languages.

MSL Considerations

ASMA machine instruction recognition and construction is completely driven by the MSL file selected by the `--target` and the `--cpu` command line options. Only machine instructions defined for the selected CPU in the MSL file will be recognized by ASMA.

ASMA Limitation: ASMA has no ability to support optional instruction operands. If an assembler instruction operand is optional, it will become required in the MSL and any source statements omitting the operand will fail assembly.

This limitation only explicitly impacts a few modern floating point instructions using the RRF format. These have optional assembler operands. When used within ASMA, the optional operand must be coded with a value of zero.

This limitation also influences MSL instruction definition and selection. For example, some legacy instructions are classified as storage-immediate. However, the immediate operand is

normally not coded because it is ignored by the instruction. Omitting the operand in legacy assemblers causes the immediate field to contain binary zeros. For these instructions a different format in MSL is used. The `LOAD PSW` instruction is an example of this. Modern usage identifies a different format, the storage format for `LOAD PSW`.

ASMA Input Statements

ASMA input statements map into five elements in sequence as illustrated by this model:

```
LABEL    OPERATION  OPERANDS        COMMENTS     \
```

The only required element is the `OPERATION` field. `COMMENTS` are always optional. Whether a `LABEL` or one or more `OPERANDS` are required or optional is dictated by the `OPERATION` field content. If the statement is continued the last character immediately preceding the end-of-line sequence must be a backslash, `\`. If `OPERANDS` are not defined for an operation, they are considered to be comments. If a `LABEL` is omitted or in the rare case prohibited, the required `OPERATION` field must be preceded by at least one space. If a `LABEL` is present it must begin in the statement.

A statement following a continued statement must contain a space in positions 1-15. A continued statement may itself be continued. Content in the continued statement beginning with position 16 logically continues the `OPERANDS` of the preceding statement.

Two exceptions to this format are supported for comment statements. If the statement begins with either an asterisk, `*`, or a period followed by an asterisk, `.*`, the entire statement is free-form. Comment statements do not recognize continuation. A comment statement starting with a period in a macro definition are considered silent comments and are not treated as model statements.

Names

Names are used within ASMA in various contexts. ASMA recognizes four different forms of names:

- Label symbol,
- Symbolic variable,
- Sequence symbols, and
- Label symbol terminated with a symbolic variable.

For assembler labels, the following attributes are supported: `D`, `I`, `L`, `M`, `O`, `S`, and `T`.

For symbolic variables and macro parameters, the following attributes are supported: `K`, `N`, and `T`.

`SETC` symbolic variables and macro parameters assigned a value of a valid label are additionally treated as labels.

ASMA Specific Behavior: For assembler labels, the following ASMA specific attribute is supported: `M`.

The following table summarizes ASMA support of attributes.

Attribute	Default	Label	Macro Parameter	Symbolic Variable	Description
`D`	`0`	yes	yes *	SETC *	Identifies whether an assembly label is defined
`I`	`0`	yes	yes *	SETC *	Number of integer supported by a constant
`K`	`0`	ERROR	yes	yes	Character count of a symbolic variable or macro parameter or sublist
`L`	`1`	yes	yes *	SETC *	Length of the area associated with a label
`M`	`0`	yes	yes *	SETC *	Relative position of a CSECT or REGION in an image file
`N`	`0`	ERROR	yes	yes	Number of elements in a parameter sublist or subscripts in a symbolic variable.
`O`	`U`	yes	yes *	SETC *	Operation associated with a name
`S`	`0`	yes	yes **	yes **	Number of digits to the right of a nominal value's decimal point
`T`	`U`	yes	yes **	yes **	Type of label or macro parameter

* For this attribute if the macro parameter or SETC symbol is assigned a label as its value, the attribute of the label is returned. When the attribute is applied to a SETA or SETB symbol an error occurs.

** See the attribute description for details.

D' Attribute

The `D` attribute identifies whether the assembler label is defined at the point the attribute is requested. A value of zero indicates the label is not defined. A value of 1 indicates it is defined. When applied to a macro parameter or `SETC` symbolic variable, the character value of the symbol is used as the basis for the identification. When applied to a `SETA` or `SETB` symbolic variable an error is recognized.

I' Attribute

The `I` attribute identifies the number of positions to the left of the implied decimal point supported by the defined packed or zoned constant. The label's explicit or implicit length determines this value, not any digits in the nominal value. All other constants report a value of 0. When applied to a macro parameter or `SETC` symbolic variable, the character value of the symbol is used as the basis for the identification. When applied to a `SETA` or `SETB` symbolic variable an error is recognized.

K' Attribute

The K attribute identifies the number of characters assigned to a symbolic variable. If characters are not assigned to the symbolic variable, the attribute is zero, as is the case for arithmetic or binary symbols. When applied to normal assembler labels an error is recognized.

L' Attribute

The L attribute identifies the length in bytes of an assembly label. For control sections, dummy sections, regions and the image file, the reported length is the total length of the corresponding label. When applied to a macro parameter or SETC symbolic variable, the character value of the symbol is used as the basis for the identification. When applied to a SETA or SETB symbolic variable an error is recognized.

M' Attribute

The M attribute identifies the CSECT or REGION position, in bytes relative to the start of the image file. All other labels report a relative position of 0. When applied to a macro parameter or SETC symbolic variable, the character value of the symbol is used as the basis for the identification. When applied to a SETA or SETB symbolic variable an error is recognized.

ASMA Specific Behavior: The M attribute is specific to ASMA.

N' Attribute

The N attribute identifies the number of elements available in a subscripted symbolic variable or macro parameter sublist. When applied to normal assembler labels an error is recognized.

O' Attribute

The operation associated with a name is identified by the O attribute. When applied to a macro parameter or a SETC symbolic variable, the character value of the symbol is used as the basis for the identification. When encountered in arithmetic expressions, the value is treated as a character self defining term. Assembly operation identification rules always apply to the operation. ASMA supports the following operation attribute values.

O' Value	Label	Macro Parameter	Symbolic Variable	Description
A	yes	yes	SETC	ASMA assembler directive
E	yes	yes	SETC	Extended mnemonic as defined by the instruction's MSL entry
M	yes	yes	SETC	In-line macro definition
O	yes	yes	SETC	MSL defined machine instruction
S	yes	yes	SETC	Macro library defined macro
U	yes	yes	SETC	Undefined, unknown or unassigned

A library defined macro that is referenced for its O attribute that has already been implicitly defined will result in an `O` attribute of `M`. The same macro that has not been explicitly defined will result in an `O` attribute of `S`.

When applied to a `SETA` or `SETB` symbolic variable an error is recognized.

S' Attribute

The number of decimal positions right of the decimal point are defined for a label. The value is determined by the first nominal value of a packed or zoned constant. All other constants result in a scale of 0. For `SETC` symbols or macro parameters whose value is a label, the `S'` attribute of the label is used. Application of the S attribute to `SETA` or `SETB` symbolic variables results in a value of 0

T' Attribute

The `T` attribute identifies the type of label or macro parameter. When encountered in arithmetic expressions, the value is treated as a character self defining term. ASMA supports the following type attribute values.

T' Value	Label	Macro Parameter	Symbolic Variable	Description
A	yes	Note 1	Note 1	A or AD address constant, aligned, implied length. See Note 3.
B	yes	Note 1	Note 1	B Binary constant
C	yes	Note 1	Note 1	C, CA, or CE character constant
D	yes	Note 1	Note 1	ASMA D quadword fixed point or long decimal-floating-point constant, aligned, implied length
E	yes	Note 1	Note 1	E short decimal-floating-point constant, aligned, implied length
F	yes	Note 1	Note 1	F fullword fixed point constant, aligned, implied length
H	yes	Note 1	Note 1	H halfword fixed point constant, aligned, implied length
I	yes	Note 1	Note 1	Machine instruction
J	yes	Note 1	Note 1	CSECT name
L	yes	Note 1	Note 1	L extended decimal-floating-point constant, aligned, implied length
N	no	yes	yes	SETA, SETB or a macro parameter or SETC symbol assigned to a self-defining term. See Note 1.
O	no	yes	no	Omitted macro parameter. See Note 1.
P	yes	Note 1	Note 1	Packed decimal constant
R	yes	Note 1	Note 1	A, S, SY or Y address constant, explicit length. See Note 3.
S	yes	Note 1	Note 1	S or SY address, aligned, implied length
U	yes	yes	yes	Undefined, unknown or unassigned. See Note 2.

T' Value	Label	Macro Parameter	Symbolic Variable	Description
W	yes	Note 1	Note 1	CCW, CCW0 or CCW1 assembler directive
X	yes	Note 1	Note 1	Hexadecimal constant
Y	yes	Note 1	Note 1	Y address constant, aligned, implied length
Z	yes	Note 1	Note 1	Zoned decimal constant
1*	yes	Note 1	Note 1	Image name
2*	yes	Note 1	Note 1	Region name
3*	yes	Note 1	Note 1	PSW, PSWS, PSW360, PSW67, PSWBC, PSWEC, PSW380, PSWXA, PSWE370, PSWE390, or PSWZ assembler directive
4*	yes	Note 1	Note 1	DSECT name

* ASMA specific values used by ASMA extensions.

Note 1 – When the macro parameter or SETC symbolic variable is assigned the value of a label, the T attribute of the label is used. If the macro parameter or SETC symbolic variable is assigned a character string value that is a self-defining term, its T attribute is N. The T attribute of an omitted macro parameter is O. If the symbolic variable is a SETA or SETB symbol, its T attribute is N.

Note 2 – The value of U may occur in these cases:

- The macro parameter or symbolic variable is *undefined*.
- The macro parameter or SETC symbolic variable is neither a self-defining term nor a valid assembler label so it is *unknown*.
- The label to which the macro parameter or SETC symbolic variable is assigned is itself *undefined* or its T attribute is *unknown* (no corresponding attribute value for its type).

Note 3 – In these cases an examination of the label's L (length) attribute is required to determine is extended type.

Labels

A label begins with any alphabetic character in upper or lower case ('A' through 'Z' and 'a' through 'z'), or any of three special characters ('$', '@' or underscore, '_') optionally followed by any of these characters or a number '0' through '9'. Labels may not begin with a number.

Labels in the LABEL field define symbols as address locations or specific values.

Labels in the OPERATION field identify machine instruction mnemonics, assembler or macro directives, previously defined macros, or operation synonyms defined by the OPSYN directive.

A label in the OPERAND field references a label defined in the LABEL field.

Symbolic Variables

Symbolic variables are labels preceded by an ampersand, '`&`'. Symbolic variables may only be used in macro language statements within a macro definition. Within macro model statements, symbolic variables may occur alone or following a label within the model's `LABEL` or `OPERATION` field and anywhere in the model statement's `OPERAND` field. Refer to the Macro Language section for other uses of symbolic variables.

ASMA Limitation: Symbolic variables will cause an error if occurring in Assembly Language statements.

Sequence Symbols

Sequence symbols are labels preceded by a period, '`.`'. Sequence symbols may only occur in Macro Language directives in the `LABEL` field or referenced in specific directives' `OPERAND` field.

ASMA Limitation: Sequence symbols may not be used outside of macro definitions.

Self-Defining Terms

ASMA input statements may contain self-defining terms. Self-defining terms define unsigned integers used alone or within expressions.

Self-defining terms look like assembler `DC` directive operands but function differently. They are utilized to define values within the assembly itself and never inherently define storage or address locations. They may be used in contexts that do.

Self-defining terms when used as the value of a prototype keyword parameter or `SETC` symbol result in the string being allowed in contexts that require a numeric value.

Binary

A binary self-defining term starts the an upper or lower case '`B`', followed by a single quote, a value composed of only a zero, '`0`' or one, '`1`' and ending with a single quote.

Example: `B'01001'` – value 9

Character

A character self-defining term starts with an upper or lower case '`C`', '`CA`' or '`CE`', followed by a single quote, a value composed of one to four characters and ending with a single quote. The numeric value is defined by the ASCII or EBCDIC code point of the character value. A '`C`' implies the EBCDIC code point. A '`CA`' explicitly uses the ASCII code point. A '`CE`' explicitly uses an EBCDIC code point. The code points are combined into a single unsigned absolute value. Two successive single quotation marks are treated as a one single quotation mark.

Example: `C'0'` – value 240 or hexadecimal 0xF0.

Example: `CA'0'` – value 48 or hexadecimal 0x30.

Example: `C'ABCD'` – value 3,250,766,788 or hexadecimal 0xC1C2C3C4.

Decimal

A decimal self-defining term is composed of a sequence of one or more numbers '`0`' through '`9`'. The term must not start with a sign.

Example: `920` – value 920 or hexadecimal 0x398.

The presence of a sign in conjunction with a self-defining term implies an expression and is only valid in contexts supporting expressions.

Hexadecimal

A hexadecimal self-defining term is composed of an upper or lower case `X`, followed by a single quote, a value of one or more hexadecimal digits, '`0`' through '`9`', '`A`' through '`F`' or '`a`' through '`f`' and ending with a single quote.

Example: `X'0Bad'` – value 2,989.

Quoted Strings

Quoted strings are surrounded by single quotation marks. Within quoted strings an ampersand or a single quote require two consecutive ampersands or single quotes to result in a single quote or ampersand within the quoted string. Quoted strings may be of any length. Quoted strings may be used in `DC` assembler directives for `C`, `CA` or `CE` type constants, default values assigned to keywords by a macro prototype statement or parameter values assigned when a macro is invoked.

Literals

A literal provides constant definition separate from a `DC` directive. A literal specification is initiated with a leading equal, `=`, sign followed by a the content of any supported `DC` operand. The actual definition of the literal is provided later in the assembly by a `LTORG` directive, or in the absence of an `LTORG` directive by the `END` directive. The literal specification is restricted to one `DC` operand but may contain multiple nominal values. Duplication factors must evaluate to values greater than zero.

See the `DC` directive description for more details.

ASMA Limitation: Literal specifications are limited to machine instruction storage operands, the target of an `ORG` directive, the address argument of a `CCW` directive and macro model statements. Literal specifications may not be used in macro directives and arithmetic expressions, nor may attributes of the literal be explicitly accessed in any context. Implicitly, a literal's length attribute is accessed when the literal specification occurs in the first operand of

a storage-to-storage instruction.

Case Sensitivity

By default the assembler is case insensitive.

The command line argument `--case` enables treatment of assembler labels, macro symbolic variables and sequence symbols with case sensitivity. All other input remains case insensitive.

Assembler statement operation field is always case insensitive. Storage allocation and defined constant types and an associated length modifier are always case insensitive.

Depending upon the use or absence of the `--case` argument, values may appear as they are in statement input or with strictly upper case letters. Internally case insensitive data is converted to upper case. This is most obvious within the symbol cross-reference portion of the assembler listing. When case sensitivity is disabled, all symbols are listed in upper-case.

Operand Values

Operand values may be either an integer or bound to an address. All operand values are derived from a mathematical expression. The expression may be simply a numeric value or derived from a calculation.

Calculations between values are allowed as follows:

Operation	Left operand	Right operand	Result
+ (addition)	integer	integer	integer
+	integer	address	address
+	address	address	prohibited
+	address	integer (Note 4)	address
- (subtraction)	integer	integer	integer
-	integer	address	prohibited
-	address	address	integer (Note 1)
-	address	Integer	address
* (multiplication)	integer	integer	integer
*	integer	address	prohibited
*	address	address	prohibited
*	address	integer	prohibited
/ (division)	integer	integer (Note 3)	integer (Note 2)
/	integer	address	prohibited

Operation	Left operand	Right operand	Result
/	address	address	prohibited
/	address	integer	prohibited

Note 1: Subtraction only allowed between addresses of the same section.

Note 2: Remainder from a division operation is discarded.

Note 3: Division by zero results in a value of zero.

Note 4: **ASMA Specific Behavior**: Dummy section relative addresses are treated as integers in this context. The integer value is the symbol's location within the dummy section.

ASMA Specific Behavior: Expression results and intermediate values are not limited to values within the range -2^{31} to $+2^{31}-1$. No actual size limit applies to calculated values. This results from **Python**'s implementation of integers that has unlimited precision:

https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex

Location Counters

ASMA Specific Behavior: Everything in this section dealing with location counters is unique to ASMA. ASMA location counter management is both similar and dissimilar to all other legacy and modern mainframe assemblers.

Similar to legacy and dissimilar to modern mainframe assemblers, the user is unable to explicitly manage location counters via an assembler directive, for example, `LOCTR`. Similar to modern assemblers and dissimilar to legacy mainframe assemblers, multiple location counters are used. Whenever a control section or dummy section is initiated it is assigned its own location counter. During pass 1, section relative addresses are utilized. During pass 2, control section relative addresses are assigned an absolute address based upon the region's starting absolute address and the location of the control section within the region. This occurs during the implicit linking process at the end of Pass 2 when the stand-alone image is created.

This processing of location counters has the effect of altering what one would find in a legacy mainframe assembler address column within the listing. Because the listing is produced based upon the control section's absolute region address, it may be logically contiguous with the preceding control section or not. By managing region start addresses and control section assignments, address management may be managed largely similar to that of a modern mainframe assembler providing location counter related assembler directives.

ASMA location counter management is most similar to the GNU `as` assembler, although the GNU `as` assembler does not allow absolute address assignment to control sections. That process is managed during linkage processing by the GNU `ld` linkage editor via linkage editor scripts, albeit only within the context of the creation of Executable and Linking Format (ELF) executable files.

Machine Instructions

Instruction recognition is defined external to the assembler by means of a text file specified explicitly in the command line or implicitly by the targeted architecture. The file is coded using the Machine Specification Language (MSL). Only central processing units (CPU) defined in the file can be a target for the assembler. Only instructions identified as valid for a target CPU by the MSL file are recognized by the assembler.

Assembler Directives

ASMA assembler directives are documented here. The following conventions are used in the directive descriptions. Anything enclosed between '<' and '>' is required variable statement content. Anything enclosed between '[' and ']' is optional.

Anytime the optional `[label]` field is identified in a statement's description it means an optional label, if provided, will be assigned the current statement's address within the active control or dummy section. Any other action for the label field content will be explicitly identified.

Anything in `UPPER CASE` is required as specified in the description.

ASMA Specific Behavior: The following directives are unique to ASMA and are not found in other assemblers:

> `ATRACEOFF`, `ATRACEON`, `PSW`, `PSWS`, `PSW360`, `PSW67`, `PSWBC`, `PSWEC`, `PSW380`, `PSWXA`, `PSWE370`, `PSWE390`, `PSWZ`, `REGION`, `XMODE`.

The following directives diverge in some respect from legacy usage. See their respective descriptions for details.

> `AMODE`, `CCW`, `DC`, `DS`, `END`, `ENTRY`, `OPSYN`, `RMODE`, `START`.

ASMA Limitation: Any assembler directive not described in this manual is not supported by ASMA.

AMODE

```
[label] AMODE <24|31|64|ANY|ANY31|ANY64>
```

The `AMODE` directive defines the addressing mode of the control section identified in the `label` field. Omitting the `label` field targets the unnamed control section.

ASMA Limitation: The `AMODE` directive is supported strictly for assembler source compatibility with other assemblers and is treated as a comment. No validation or processing occurs for either the label or operand field.

ATRACEOFF

```
[label] ATRACEOFF [oper][,oper]...
```

The ATRACEOFF directive disables operational tracing previously enabled via the --oper command line argument or a ATRACEON assembler directive. An optional [label], if provided, is ignored. Tracing is disabled for each of the supplied operations identified by [oper].

No action is taken if the operation is invalid, tracing has not been enabled for an operation, the operation is a defined macro, or no operations are supplied in the operand field. Use the MHELP directive to alter the tracing status of defined macros.

The ATRACEOFF and ATRACEON directives provide fine grain control of internal tracing not available with the --oper command-line option alone.

ATRACEON

```
[label] ATRACEOFF [oper][,oper]...
```

The ATRACEON directive enables internal tracing for one or more operations. An optional [label], if provided, is ignored. Tracing is enabled for each of the supplied operations identified by [oper].

No action is taken if the operation is invalid or tracing is already enabled for the operation, or the operation is a defined macro. If no operations are supplied in the operand field, the current table of instructions for which tracing has been enabled is displayed. Use the MHELP directive to alter the tracing status of defined macros.

The ATRACEOFF and ATRACEON directives provide fine grain control of internal tracing not available with the --oper command-line option alone.

CCW

```
[label] CCW   <command>,<address>,<flags>,<count>
```

The CCW directive constructs a Channel Command Word based upon the current execution mode format setting for a CCW. If the CCW execution mode has been set to 'none', the CCW directive is not recognized. See the XMODE directive.

ASMA Specific Behavior: The legacy operation of the CCW directive always created a Format-0 CCW for backward compatibility with source predating the existence of the Format-1 CCW. The above description of the CCW directive diverges from this legacy behavior.

The following priority exists for channel command word format generation:

1. An explicit `CCW0` or `CCW1` directive in the program source. An explicit `CCW0` or `CCW1` directive ignores the current `XMODE CCW` setting.
2. The current setting by an explicit `XMODE CCW` directive at the time a `CCW` directive is encountered. An explicit `XMODE CCW` directive overrides the `--ccw` command line argument.
3. A value supplied by the ASMA command-line option `--ccw` establishing the `XMODE CCW` setting preceding an explicit `XMODE CCW` directive. A command line argument overrides the MSL file.
4. The value identified by the MSL file selected by means of the `--target` or `--cpu` ASMA command line argument.

The current setting is made available to a macro by means of the system variable symbol `&SYSCCW`.

CCW0

```
[label] CCW0  <command>,<address>,<flags>,<count>
```

The `CCW0` directive generate a Format-0 Channel Command Word. The channel command word will be double word aligned. All operands are expressions. The `<command>`, `<flags>`, and `<count>` operands must evaluate to an integer. The `<address>` operand must evaluate to an address and may contain a literal specification. All operands are required. All reserved bits are set to zero. Use `DC` directives to build an invalid Format-0 Channel Command Word for testing purposes.

CCW1

```
[label] CCW1  <command>,<address>,<flags>,<count>
```

The `CCW1` directive generates a Format-1 Channel Command Word. All of the operand rules are the same as for the `CCW1` directive. All reserved bits are set to zero. Use `DC` directives to build an invalid Format-1 Channel Command Word for testing purposes.

CNOP

```
[label]  CNOP  <byte>,<boundary>    [comments]
```

The CNOP directive conditionally includes BCR 0,0 instructions, X'0700', into the assembly based upon the <byte> position within the <boundary>.

Both <byte> and <boundary> are required expressions and must evaluate to an absolute value. The <byte> must be a non-negative even, divisible by 2 without a remainder, value and less than the <boundary>. The <boundary> may evaluate to 4, 8, or 16.

The optional [label], when provided, will always be assigned a length attribute of 1.

COPY

```
[seqsym] COPY  'filename' [comments]
```

The COPY directive inserts into the input stream statements from the file identified between the single quoted string that is the single required argument of the COPY directive. Single quotation marks are prohibited within the file name. The file name itself must be compatible with the platform in use. The file is searched within the directories specified by the ASMPATH environment variable and the current active directory.

A sequence symbol may be supplied within a macro definition.

CSECT

```
[label]  CSECT [comments]
```

The CSECT directive initiates a new control section or continues a previously initiated control section. A new control section is added to the current active region. Control sections are double word aligned within the region. The label is the symbol assigned to the control section. The symbol's value is the address of the start of the control section. Its length attribute is the total length of the control section. Its image attribute is the position of the control section within the binary image relative to the image's start.

If the optional label field is omitted, an unnamed control section is created or continued. Being unnamed, attribute values are not available for an unnamed control section.

If no current active region exists, an unnamed region with a starting address of 0 is created to

which the new control section is added.

When a control section is continued, the region within which the control section is defined automatically becomes the current active region.

Any operand data that may be supplied is treated as a comment.

DC

```
[label] DC    [d]A[Ln](address[,...])[,...]
[label] DC    [d]t[Ln]'content[,...]'[,...]
```

The DC directive defines storage usage and content. Usage is specified by a usage description. Usage descriptions are identical to those used in the `DS` directive. The content is provided within a pair of single quotes or a left/right parenthesis pair depending upon the type of storage usage. The '`A`', '`AD`', '`S`', '`SY`', and '`Y`' types require a parentheses pair. All other types use the pair of single quotes. Spaces may be embedded within the quotes for readability. Multiple content values may be used where they share storage usage. Multiple definitions may be supplied, separated by a single comma. Multiple nominal values may be supplied for the content except as noted below.

The `C`, `CA` and `CE` constant types allow only a single value to be specified. Two single quotes or ampersands in succession are converted into one single quote or single ampersand within the assembled constant.

The optional duplication `[d]` factor may be either an unsigned decimal self-defining term or an expression enclosed in parenthesis. The expression must evaluate to an integer of zero or greater and may only reference previously defined labels.

The optional length modifier `[Ln]` may be either an unsigned decimal self-defining term or an expression enclosed in parenthesis. The value must not be less than one nor exceed the maximum allowed for the constant type and not excluded for the type. The initial required `L` may be either upper or lower case. The expression may only reference previously defined labels.

This table describes various attributes of the usage definition and content specification. “Signed” implies that a numeric value is preceded by either a plus sign, +, or minus sign, -. If either sign is omitted, a plus sign is assumed. “Unsigned' means that a numeric value is preceded by the letter `U`.

Type	Implied Length	Implied Alignment	Trunc / Pad	Image Content and Nominal Value
`A`	4	4	Left	An expression evaluating to an address or integer >=0

Type	Implied Length	Implied Alignment	Trunc / Pad	Image Content and Nominal Value
AD	8	8	Left	An expression evaluating to an address or integer >=0
B	content	none	Left	Binary data defined by '0' and '1'
C	content	none	Right	EBCDIC character data translated from ASCII input
CA	content	none	Right	ASCII character data as allowed by **Python** UTF-8 encoding
CE	content	none	Right	EBCDIC character data translated from ASCII input
D	8	8	Left	Signed or unsigned integer (but not as a floating point constant)
DD	8	8	none	Signed or unsigned long decimal-floating-point value
ED	4	4	none	Signed or unsigned short decimal-floating-point value
F	4	4	Left	Signed or unsigned integer
FD	8	8	Left	Signed or unsigned integer
H	2	2	Left	Signed or unsigned integer
LD	16	8	none	Signed or unsigned extended decimal-floating-point value
P	content	none	Left	Signed or unsigned packed decimal data, decimal point optional
S	2	2	none	An expression evaluating to an address in base and 12-bit displacement format
SY	3	none	none	An expression evaluating to an address in base and 20-bit displacement format
X	content	none	Left	Hexadecimal data defined by hex digits 0-9 and A-F
Y	2	2	Left	An expression evaluating to an address or integer of >= 0
Z	content	none	Right	Signed or unsigned zone decimal data, decimal point optional

If the optional [label] is specified, it acquires the address of the first storage usage description and its length. A sequence symbol may be supplied in the label field within a macro definition.

ASMA Specific Behavior: The supported constant formats are a subset of those supported by legacy assemblers. Rather than defining a hexadecimal floating point value the D constant type defines an 8-byte signed integer value. It acts as a synonym for the FD constant type. This allows use of the D constant type in legacy programs for storage allocation and large integer definitions where the FD constant type was unavailable.

ASMA Specific Behavior: The plus and minus signs are always treated as expression operators. If used within a duplication factor or length modifier, the expression must be enclosed in parenthesis. For example, a duplication factor of +3 must be coded as (+3).

ASMA Limitations: The following constant types are not supported by ASMA or not supported by ASMA in the same manner as legacy assemblers: CU, D, DB, DH, E, EB, EH, G, J, L, LB, LH, LQ, Q, R, RD, and V.

ASMA Limitations: Only the length modifier (`L`) is supported. The scale (`S`) and exponent (`E`) modifiers are not supported. The program type sub-field (`P`) is not supported.

ASMA Limitations: Use of floating point numeric values in other than floating point constant types is not supported.

DROP

```
[seqsym] DROP  <reg>[,reg...]
```

The `DROP` directive ensures no `USING` assignment exists for a register. At least one register must be identified. Up to 15 more optional registers may be specified. Each `<reg>` operand is an expression that must evaluates to an integer between 0 and 15, inclusive.

An optional `[seqsym]` symbol may be supplied in the label field within a macro definition.

DS

```
[label]  DS  [d]t[Ln][,...]
```

The `DS` directive defines storage usage. Usage is specified by a description operand. A description operand includes an optional duplication factor, `[d]`, a required type, `t`, and an optional explicit length, `[Ln]`. Nominal values are allowed and if present influence the amount of storage defined. If an explicit length is specified, any implied alignment of the type is suppressed. Multiple storage usage operands are allowed.

If the optional `[label]` is specified it acquires the address of the first storage usage description and the length of the storage usage definitions taken as a whole. A sequence symbol may be supplied in the label field within a macro definition.

See the `DC` directive description of supported types and implied lengths and alignment.

Within the created image data, areas defined by the `DS` directive will initialize each storage usage to binary zeros. Other content can replace the binary zero content by use of the `ORG` directive to overlay the default content.

DSECT

```
<label> DSECT [comments]
```

The DSECT directive initiates or continues a previously initiated dummy section. The label field identifies the symbol associated with the dummy section. The value will always be a section relative address of zero. The label's length will be the total length of the dummy section.

Any operand information provided is treated as a comment.

EJECT

```
[seqsym] EJECT [comments]
```

The `EJECT` directive introduces a new page in the assembly listing. An optional `[seqsym]` may be provided in the label field within a macro definition. Any operand field information is treated as comments.

END

```
[seqsym] END   [entry]
```

The `END` directive terminates the assembly. Once encountered, no more input statements are allowed although pending literal specifications are defined following the `END` directive. The optional `[seqsym]` field may be provided within a macro definition.

The optional `[entry]` operand defines the entry address of the image.

ASMA Specific Behavior: The `ENTRY` directive is an alternative method for defining the image entry point. See the `ENTRY` directive.

ENTRY

```
[seqsym] ENTRY <address>
```

The `ENTRY` directive defines the entry address of the program within the image. Because the output image file has no inherent structure, no mechanism exists for the communication of the entry point to any actual user of the image file contents. It is reported for use in whatever downstream process would require this information.

The `<address>` operand is required and may be an expression. The value to which the expression evaluates is reported as the entry point.

An ENTRY directive supersedes any previous ENTRY directive within the assembly. Only the last ENTRY directive is used.

An optional [seqsym] may be supplied in the label field within a macro definition.

ASMA Specific Behavior: The usage of the ENTRY directive to define a program entry point instead of an entry from an externally linked program diverges from legacy behavior.

EQU

```
<label> EQU   <expression>[,length]
```

The EQU directive assigns a value to the label. The value may be an address or an integer depending on the evaluation of the required expression.

An optional operand, length, may be supplied. The value of the length operand expression specifies the length attribute of the symbol defined by the EQU directive. The length operand must evaluate to an integer. If omitted, the length attribute of the symbol defaults to one.

LTORG

```
[label] LTORG [comment]
```

The LTORG directive initiates the definition of the currently pending literal specifications. Specifications are grouped by size (largest to smallest) aligned to 8-, 4-, or 2-byte boundaries as specified explicitly or implicitly by the literal specification. All other sizes follow the aligned literal definitions on unaligned locations.

If the optional label is provided, it is assigned to the location of the start of the literal pool definitions, or the current location if no definitions are pending. The label will have a length attribute of 1 and a type attribute of 'U'.

MHELP

```
[seqsym] MHELP <action> [comment]
```

Establishes macro diagnostic information. The required <action> operand is an arithmetic expression composed of binary, decimal or hexadecimal self-defining terms. The <action> operand identifies a set of macro language trace, dump and call controls to be enabled until

the next MHELP directive is encountered. An optional [seqsym] may be provided in the label field within a macro definition.

Values above 255 control the maximum &SYSNDX value allowed during the assembly. If none of the bits in the next to the last byte of the value are set, &SYSNDX value monitoring is suppressed.

Other trace and dump options and their respective values are described in this table. Only SET symbols defined without subscripts are dumped.

Binary	Decimal	Description
B'00000001'	1	Call trace of macro name, nesting depth and &SYSNDX
B'00000010'	2	Branch trace for AGO and AIF macro directives
B'00000100'	4	Dump SET symbols before AIF executed
B'00001000'	8	Dump SET symbols when MEND or MEXIT encountered
B'00010000'	16	Dump macro parameter values upon macro entry
B'00100000'	32	Suppress dumping global SET symbols with action 4 or 8
B'01000000'	64	Include hexadecimal values of SETC symbols with actions 4, 8 or 16. See below.
B'10000000'	128	Suppresses currently active MHELP actions.

ASMA Specific Behavior: The <action> operand is not limited to a single decimal or binary self-defining term. Macro trace and dump information is sent to the ASMA system output file, not the assembler listing. Redirect the system output to a file if excessive information is generated.

ASMA Specific Behavior: SETC variable symbols support both ASCII and EBCDIC values. When action 64 is used, the EBCDIC hexadecimal values are included in the dump along with the ASCII characters. Depending upon the context, the originating ASCII or translated EBCDIC value will be used. See the "SETC" macro language section for details.

ASMA Specific Behavior: Action 2, branch tracing, is not limited to in-line macro definitions. Macro definitions from a macro library may also be traced. The line numbers provided in the trace information relate to the source of the macro definition. In-line macro definitions trace the line number provided by the assembly listing. Macro library definitions trace the line numbers of the defining macro library file. When tracing is used with in-line macro definitions, PRINT OFF must not be in effect.

MNOTE

```
[seqsym] MNOTE <severity>,<message>
```

The MNOTE directive generates an in-line message. The <severity> operand is required. It may be either a numeric value or an asterisk, '*'. The message operand is required and is coded within quotes. The optional [seqsym] field may be supplied in a macro definition.

ASMA Limitation: The severity operand is supported for compatibility. ASMA does not support severity processing. A severity of '*' **is** treated as an informational message. All numeric severity's

OPSYN

```
     <label> OPSYN [oper]
or           OPSYN
```

The OPSYN directive defines, redefines or deletes an operation. The required <label> field identifies the operation being defined, redefined or deleted. If the single optional [oper] operand is present, it identifies the machine instruction, assembler directive or macro for which the <label> becomes a synonym. If the [oper] operand is omitted, the <label> field identifies the synonym, assembler directive, machine instruction or macro being deleted. Deleted synonyms no longer provide access to the underlying assembler directive, machine instruction or macro.

The optional [oper] operand when provided follows the same rules of construction as do labels.

Comments are allowed in an OPSYN statement deleting a synonym if the operand field is started by a comma.

The OPSYN directive associates the existing processing of a machine instruction, assembler directive, macro or other synonym with a new operation name. Machine instructions and assembler directives exist because the processing is part of the assembler.

For macros, the processing logic is associated with the macro's definition. To associate a macro with a different name requires it to be defined previous to the OPSYN directive providing its synonym. The use of the name in the OPSYN directive does not cause the macro to be defined if it resides in the MACLIB environment variable's path. To use OPSYN with an externally defined macro requires the macro to be defined by being accessed via a COPY directive prior to the use of the OPSYN directive. This requires the directory in which the macro exists to be part of the ASMPATH environment variable. The same directory may also reside in the MACLIB environment variable.

When a macro overrides, thereby replacing, a machine instruction or assembler directive by virtue of its name being the same, access to the original operation may be restored by these

steps. In the example a macro named `SVC` will replace the instruction.

1. Prior to defining the macro with the same name, create a synonym to the original operation, for example: `SAVESVC OPSYN SVC`
2. Define the replacement macro either inline or by using a COPY directive (otherwise the macro definition will not occur because a defined instruction takes precedence over an undefined macro of the same name).
3. Restore access to the original operation name by defining the original name as a synonym for the saved synonym defined in step 1 (and eliminating access to the replacement macro), for example: `SVC OPSYN SAVESVC`

ASMA Specific Behavior: Access to an underlying assembler directive, machine instruction or macro is still possible by use of its original name. The synonym provides simply an alternative name by which to access the underlying operation.

ASMA Specific Behavior: If neither a label nor operand are supplied, the current operation synonym table is printed on SYSOUT. The output includes the operation accessed by the synonym, its `O'` attribute and additional information useful for debugging.

ORG

```
[label] ORG   <expression>
```

The `ORG` directive assigns a new value to the current location counter, represented by '`*`' within expressions. The required `<expression>` operand must evaluate to an address or a literal specification within the current active control section.

If the optional `label` is provided it is assigned the value of the current location counter before the new location is assigned to it. An optional `[seqsym]` may be provided in the label field within a macro definition.

POP

```
[seqsym] POP option[,NOPRINT]
```

The `POP` directive restores the most recently saved setting of the specified option or options. Options are treated in a case insensitive manner. Option settings are saved by use of the `PUSH` directive.

Supported options:

- `PRINT` – Identifies the saved status of the PRINT options to be restored.

- NOPRINT – If supplied, causes the printing of the POP directive to be suppressed in the listing. Otherwise the current setting of the PRINT ON or PRINT OFF option applies to the statement. If provided, NOPRINT must be the last supplied option.

An optional [seqsym] may be supplied in the label field within a macro definition.

ASMA Limitation: Only the PRINT option of the first operand is supported for the POP directive. Other options for example, USING or ACONTROL are not supported.

PRINT

```
[seqsym] PRINT <option>[,option]...[,NOPRINT]
```

The PRINT directive provides listing control settings. Each operand specifies a case insensitive option. At least one option is required. The following options are supported:

- ON – Enables generation of assembler statements in the listing.
- OFF – Disables assembler statements in the listing.
- GEN – Enables macro generated statements in the listing.
- NOGEN – Disables macro generated statements in the listing.
- DATA – Causes all object code generated by a statement in the listing.
- NODATA – Causes a maximum of eight bytes of object code generated by a statement in the listing.
- NOPRINT – Causes the printing of the PRINT directive itself to be suppressed. Otherwise the directive is printed in the listing regardless of the current setting of PRINT ON or PRINT OFF. If provided it must be the last option.

At the start of the assembly the following options are in effect: ON, GEN, and NODATA.

An optional [seqsym] may be supplied in the label field within a macro definition.

PSW

```
[label] PSW   <sys>,<key>,<a|amwp|mwp>,<prog>,<addr>[,amode]
```

The PSW directive constructs a Program Status Word based upon the current execution mode format setting for a PSW. If the PSW execution mode has been set to 'none', the PSW directive is not recognized. See the XMODE directive.

The following priority exists for Program Status Word format generation:

1. An explicit `PSWxx` directive in the program source. An explicit `PSWxx` directive ignores the current XMODE PSW setting.
2. The current setting by an explicit `XMODE PSW` directive at the time a `PSW` directive is encountered. An explicit `XMODE PSW` directive overrides the `--psw` command line argument.
3. A value supplied by the ASMA command-line option `--psw` establishing the XMODE PSW setting preceding an explicit `XMODE PSW` directive. A command line argument overrides the MSL file.
4. The value identified by the MSL file selected by means of the `--target` or `--cpu` ASMA command line argument.

The current setting is made available to a macro by means of the system variable symbol `&SYSPSW`.

An optional `[label]` is assigned the location of the PSW. An optional `[seqsym]` may be supplied in the label field within a macro definition.

PSWS

```
[label]  PSWS  <sys>,<key>,<a>,<prog>,<addr>[,amode]
```

The `PSWS` directive defines a 32-bit short Program Status Word used on S/360 Model 20. The PSW is not aligned because no alignment requirement exists for S/360 Model 20 PSW's. The label is optional. Five operands are required. Each operand is an expression. The following table describes the operands.

Operand	Identification	PSW Bits	Description
1	`<sys>`	7	Channel mask
2	`<key>`	Not Used	Note 1.
3	`<a>`	6	ASCII Mode bit
4	`<prog>`	2, 3	Condition Code
5	`<addr>`	16 - 31	Instruction address or integer
6	`[amode]`	Not used	Note 2.

Note 1: The `key` operand is not used by `PSWS`. It must be syntactically correct, but is otherwise ignored. It is supported for compatibility with other PSW related directives.

Note 2: The `amode` operand is not used by `PSWS`. If specified it must be syntactically correct but is otherwise ignored. It is supported for compatibility with other PSW related directives.

All reserved bits are set to zero and any required bit settings are enforced. Use `DC` directives to build an invalid PSW for testing purposes.

An optional `[label]` is assigned the location of the PSW. An optional `[seqsym]` may be supplied in the label field within a macro definition.

PSW360

```
[label]  PSW360  <sys>,<key>,<amwp>,<prog>,<addr>[,amode]
```

The `PSW360` directive defines a 64-bit Program Status Word used on all S/360 models other than model 20. The PSW is double word aligned. The label is optional. Five operands are required and the last operand, `amode`, is optional. Each operand is an expression. The following table describes the operands.

Operand	Identification	PSW Bits	Description
1	`<sys>`	0 - 7	Interrupt masks
2	`<key>`	8 - 11	Storage protection key
3	`<amwp>`	12 -15	Note 1.
4	`<prog>`	34 -39	Condition Code and program mask
5	`<addr>`	40 - 63	Instruction address
6	`<amode>`	Not used	Note 2.

Note 1: This operand sets the following PSW bits:

- bit 12 – ASCII mode bit,
- bit 13 – Machine-check interrupt mask
- bit 14 – Wait state
- bit 15 – Problem state

Note 2: The `amode` operand is not used by `PSW360`. If specified it must be syntactically correct but is otherwise ignored. It is supported for compatibility with other PSW related directives.

All reserved bits are set to zero and any required bit settings are enforced. Use `DC` directives to build an invalid PSW for testing purposes.

An optional [label] is assigned the location of the PSW. An optional [seqsym] may be supplied in the label field within a macro definition.

PSW67

```
[label]  PSW67  <sys>,<key>,<amwp>,<prog>,<addr>[,amode]
```

The PSW67 directive creates a S/360 model 67 67-mode PSW. The PSW is double word aligned. The label is optional. Five operands are required and the last operand, amode, is optional. The last operand, amode, is optional. Each operand is an expression. The following table describes the operands.

Operand	Identification	PSW Bits	Description
1	<sys>	5 - 7	Interrupt masks and controls
2	<key>	8 - 11	Storage protection key
3	<amwp>	12 -15	Note 1.
4	<prog>	16 -23	Condition Code and program mask
5	<addr>	32 - 63	Instruction address
6	[amode]	4	Note 2.

Note 1: This operand sets the following PSW bits:

- bit 12 – ASCII mode bit,
- bit 13 – Machine-check interrupt mask
- bit 14 – Wait state
- bit 15 – Problem state

Note 2: This operand defines the address mode. As with the other operands it is an expression. If omitted, amode defaults to 0, 24-bit address mode. The following values are accepted to specify the address mode:

- 0 – 24-bit address mode,
- 1 – 32-bit address mode,
- 24 – 24-bit address mode, or
- 32 – 32-bit address mode.

All reserved bits are set to zero and any required bit settings are enforced. Use DC directives

to build an invalid PSW for testing purposes.

An optional `[label]` is assigned the location of the PSW. An optional `[seqsym]` may be supplied in the label field within a macro definition.

PSWBC

```
[label]  PSWBC <sys>,<key>,<mwp>,<prog>,<addr>[,amode]
```

The `PSWBC` directive defines a 64-bit S/370 Basic Control Program Status Word. The PSW is double word aligned. The label is optional. Five operands are required and the last operand, `amode`, is optional. Each operand is an expression. The following table describes the operands. The format bit, bit 12 is set to zero.

Operand	Identification	PSW Bits	Description
1	`<sys>`	0 - 7	Interrupt masks
2	`<key>`	8 - 11	Storage protection key
3	`<mwp>`	13 -15	Note 1.
4	`<prog>`	34 -39	Condition Code and program mask
5	`<addr>`	40 - 63	Instruction address
6	`[amode]`	Not used	Note 2.

Note 1: This operand sets the following PSW bits:

- bit 13 – Machine-check interrupt mask
- bit 14 – Wait state
- bit 15 – Problem state

Note 2: The `amode` operand is not used by `PSWBC`. If specified it must be syntactically correct but is otherwise ignored. It is supported for compatibility with other PSW related directives.

All reserved bits are set to zero and any required bit settings are enforced. Use `DC` directives to build an invalid PSW for testing purposes.

An optional `[label]` is assigned the location of the PSW. An optional `[seqsym]` may be supplied in the label field within a macro definition.

PSWEC

```
[label]  PSWEC <sys>,<key>,<mwp>,<prog>,<addr>[,amode]
```

The PSWEC directive defines a 64-bit S/370 Extended Control Mode Program Status Word. The PSW is double word aligned. The label is optional. Five operands are required and the last operand, amode, is optional. Each operand is an expression. The following table describes the operands. Not all bit settings are allowed on all processors or models. The format bit, bit 12 is set to one.

Operand	Identification	PSW Bits	Description
1	<sys>	0 - 7	Interrupt masks
2	<key>	8 - 11	Storage protection key
3	<mwp>	13 -15	Note 1.
4	<prog>	16 -23	Address space, condition code and program mask
5	<addr>	40 - 63	Instruction address
6	[amode]	Not used	Note 2.

Note 1: This operand sets the following PSW bits:

- bit 13 – Machine-check interrupt mask
- bit 14 – Wait state
- bit 15 – Problem state

Note 2: The amode operand is not used by PSWEC. If specified it must be syntactically correct but is otherwise ignored. It is supported for compatibility with other PSW related directives.

All reserved bits are set to zero and any required bit settings are enforced. Use DC directives to build an invalid PSW for testing purposes.

An optional [label] is assigned the location of the PSW. An optional [seqsym] may be supplied in the label field within a macro definition.

PSW380, PSWXA, PSWE370, PSWE390

```
[label]  PSW380  <sys>,<key>,<mwp>,<prog>,<addr>[,<amode>]
[label]  PSWXA   <sys>,<key>,<mwp>,<prog>,<addr>[,<amode>]
[label]  PSWE370 <sys>,<key>,<mwp>,<prog>,<addr>[,<amode>]
[label]  PSWE390 <sys>,<key>,<mwp>,<prog>,<addr>[,<amode>]
```

The PSW380, PSWXA, PSWE370 and PSWE390 directives define a 64-bit bimodal address mode Program Status Word used in Hercules specific S/380, 370-XA, ESA/370 and ESA/390 modes, respectively. z/Architecture ® models also support PSWE390 PSW's. The PSW is double word aligned. The label is optional. Five operands are required and the last operand, amode, is optional. Each operand is an expression. The following table describes the operands.

Operand	Identification	PSW Bits	Description
1	<sys>	0 - 7	Interrupt masks
2	<key>	8 - 11	Storage protection key
3	<mwp>	13 -15	Note 1.
4	<prog>	16 -23	Address space, condition code and program mask
5	<addr>	40 -63	Instruction address
6	[amode]	32	Note 2.

Note 1: This operand sets the following PSW bits:

- bit 13 – Machine-check interrupt mask
- bit 14 – Wait state
- bit 15 – Problem state

Note 2: This operand defines the address mode. As with the other operands it is an expression. If omitted, amode defaults to 0, 24-bit address mode. The following values are accepted to specify the address mode:

- 0 – 24-bit address mode
- 1 – 31-bit address mode
- 24 – 24-bit address mode
- 31 – 31-bit address mode

All reserved bits are set to zero and any required bit settings are enforced. Use DC directives to build an invalid PSW for testing purposes.

An optional [label] is assigned the location of the PSW. An optional [seqsym] may be supplied in the label field within a macro definition.

PSWZ

```
[label]  PSWZ  <sys>,<key>,<mwp>,<prog>,<addr>[,<amode>]
```

The `PSWZ` directive defines a 128-bit z/Architecture Program Status Word used on all z/Architecture models. The PSW is double word aligned. The label is optional. Six operands are required. Each operand is an expression. The following table describes the operands.

Operand	Identification	PSW Bits	Description
1	`<sys>`	0 - 7	Interrupt masks
2	`<key>`	8 - 11	Storage protection key
3	`<mwp>`	13 -15	Note 1.
4	`<prog>`	16 - 24	Address space, condition code and program mask
5	`<addr>`	64 -127	Instruction address
6	`[amode]`	31, 32	Note 2.

Note 1: This operand sets the following PSW bits:

- bit 13 – Machine-check interrupt mask
- bit 14 – Wait state
- bit 15 – Problem state

Note 2: This operand defines the address mode. As with the other operands it is an expression. If omitted, `amode` defaults to 0, 24-bit address mode. The following expression values are accepted to specify the address mode:

- 0 – 24-bit address mode
- 1 – 31-bit address mode
- 3 – 64-bit address mode
- 24 – 24-bit address mode
- 31 – 31-bit address mode
- 64 – 64-bit address mode.

All reserved bits are set to zero and any required bit settings are enforced. Use `DC` directives to build an invalid PSW for testing purposes.

An optional `[label]` is assigned the location of the PSW. An optional `[seqsym]` may be supplied in the label field within a macro definition.

PUSH

```
[seqsym] PUSH option[,NOPRINT]
```

The PUSH directive saves the current setting of the specified option or options. Options are treated in a case insensitive manner. Option settings are not altered by the process of saving its current setting. Option settings are restored by use of the POP directive.

Supported options:

- PRINT – Identifies the status of the PRINT options is to be saved.
- NOPRINT – If supplied, causes the printing of the PUSH directive to be suppressed in the listing. Otherwise the current setting of the PRINT ON or PRINT OFF option applies to the statement. If provided, NOPRINT must be the last supplied option.

An optional [seqsym] may be supplied in the label field within a macro definition.

ASMA Limitation: Only the PRINT option is supported for the PUSH directives first operand. Other options, for example, USING or ACONTROL are not supported.

REGION

```
[label] REGION [comments]
```

The REGION directive continues a previously initiated region. The label identifies the region that is being continued. The control section active in the region at the time it was interrupted automatically becomes the active control section following activation of the region.

If the label is omitted, the previously created unnamed region becomes the active region.

Initiation of a new region requires use of the START directive.

If any operand information is provided it is treated as a comment.

RMODE

```
[label] RMODE <24|31|64|ANY>
```

The RMODE directive defines the residency mode of the control section identified in the label field. Omitting the label field targets the unnamed control section.

ASMA Limitation: The RMODE directive is supported strictly for assembler source compatibility with other assemblers and is treated as a comment. No validation or processing

occurs for either the `label` or `operand` field.

SPACE

```
[seqsym] SPACE  [expression]
```

The `SPACE` directive inserts one or more spaces into the listing as specified by the expression operand. The expression must consist of binary, decimal or hexadecimal self-defining terms. If the expression operand is omitted, the directive defaults to one space. If the value of the expression causes the page to exceeds the number of lines per page, 55, an `EJECT` directive is implied. If supplied, the optional label is ignored.

An optional `[seqsym]` may be supplied in the label field within a macro definition.

START

```
[label]  START  [expression][,[region]]
```

The `START` directive initiates a new control section and optionally a new region. The results of the ASMA `START` directive model the effects of the first `START` directive of legacy programs within the context of address regions within an ASMA program. The optional `label` field identifies the name of the new control section. If the `label` field is omitted an unnamed control section is created. An optional `[seqsym]` may be supplied in the label field within a macro definition.

The operands to the `START` directive define the optional new region being created into which the new control section is placed. The `expression` operand defines the starting address of the region. The `region` operand identifies the new region name. If the `region` operand is present but the `expression` operand is omitted, the starting address of the new region is 0. If the `expression` operand is present but the `region` operand is omitted, a new unnamed region will be created.

If both of the `expression` and `region` operands are omitted, the new control section is placed into the current active region. If no active region exists, an unnamed region is created starting at address 0.

An error condition exists when the named or unnamed region or control section that would be created already exists.

For a `START` directive to which comments are applied but neither operand is used, a comma is required in the operand field. Otherwise the comments will be interpreted as the start of the `expression` operand.

The following table summarizes the possible cases and the actions taken by the `START` directive. The address defined by the `<expression>` operand applies to the created region. The address assigned to the control section is dictated by the placement within the region to which it is added. If it is the first control section of the region, it will have the same assigned address as the region. If it is added to a region already containing one or more control sections, the control section will start on the next doubleword following the preceding control section in the region.

<label>	<expression>	<region>	Action
omitted	omitted	omitted	Unnamed control section created in the current active region. If no active region, the unnamed region is created starting at address 0.
omitted	present	omitted	Unnamed control section created in a new unnamed region starting at the address of the expression
omitted	omitted	present	Unnamed control section created in a new named region starting at address 0.
omitted	present	present	Unnamed control section created in a new named region starting at the address of the expression
present	omitted	omitted	Named control section created in the current active region. If no active region, the unnamed region is created starting at address 0.
present	present	omitted	Named control section created in a new unnamed region starting at the address of the expression.
present	omitted	present	Named control section created in a new named region starting at address 0.
present	present	present	Named control section created in a new named region starting at the address of the expression.

ASMA Specific Behavior: ASMA supports multiple regions, each with its own independent location counter. Only the `START` directive creates a new region. Multiple `START` directives are allowed to support multiple regions. For programs written expressly for use of ASMA, it is recommend that all operands and the label field be used to minimize interplay of named and unnamed regions and sections.

ASMA Specific Behavior: The unnamed region will be assigned a file name of `unnamed.bin` when the `--gldipl` option is used. All other regions utilize their assigned name when generating list directed IPL output.

TITLE

```
[seqsym] TITLE  'New Listing Title'
```

The TITLE directive allows specification of a new title for the assembly listing. The title may contain upper or lower case letters and any other character other than a single quote. Two adjacent single quotes or ampersands are required to represent a single quote or single ampersand, respectively, in the printed listing title. An implied EJECT occurs before the new title is used. A title in excess of 85 characters is truncated on the right to 85 characters. Title characters are left justified in the the title line.

An optional [seqsym] may be provided in the label field within a macro definition.

USING

```
[seqsym] USING <address>,<reg>[,<reg>]
```

The USING directive establishes one or more base registers for the resolution of storage accesses into a base and displacement within machine instructions. The <address> and each supplied <reg> operand are expressions. The <address> expression must evaluate to an address or location within a dummy section. A <reg> expression must evaluate to an integer between 0 and 15 inclusive.

Each additional register is assigned a base value 4096 bytes from the previous assigned address.

A base register assignment in place for any of the specified registers replaces a previous assignment without the need of a DROP directive.

An optional [seqsym] may be supplied in the label field within a macro definition.

ASMA Specific Behavior: Because USING directive processing occurs during Pass 2 statement processing following the assignment of absolute address locations, a USING directive associated with a control section will result in one or more registers being associated with the absolute address. Any address validly in range for the instruction's displacement field is accepted regardless of control sections involved in the USING directive address expression, the instruction itself or an instruction's operand. Instruction operands targeting dummy section labels require the same dummy section to be associated with the USING address expression as occurs in the instruction operand.

XMODE

```
[seqsym] XMODE CCW,<0|1|none>
[seqsym] XMODE PSW,<S|360|67|BC|EC|380|XA|E370|E390|Z|none>
```

The `XMODE` directive sets the current CCW or PSW execution mode. In both cases the execution mode determines the specific PSW format or CCW format created by the `PSW` or `CCW` directive, respectively. In either case, specifying `none` removes the existing setting and causes the `PSW` or `CCW` directive to not be recognized by the assembler.

The first occurrence of the `XMODE` directive will override a command line value provided by either the `--ccw` or `--psw` option. If a command line option is not specified, the default is the expected CCW or PSW format defined for the CPU in the MSL database.

An optional `[seqsym]` may be supplied in the label field within a macro definition.

Macro Language

ASMA supports two separate languages:

- the ASMA assembler language, described previously, is primarily declarative in nature, and
- the ASMA macro language, described in this section, is primarily interpretive and results in the creation of declarative assembler language statements.

The ASMA macro language processing provides a subset of some legacy implementations and a super set of others. It should not be expected that any legacy macro definition will work without some changes to accommodate the ASMA macro language.

The macro language recognizes two types of statements. Statements that occur in open code, outside of a macro definition, and statements that occur within a macro definition. Macro Language directives may only occur within a macro definition.

Macro definitions may occur within an assembly source file, copied into an assembly using the assembler COPY directive or implicitly defined upon first use when defined within a macro library. See the next section, “Macro Libraries”, for details.

Macro Libraries

Macro definitions may be placed within a macro library. A macro library is a directory containing files that only define macros. If an operation is not a machine instruction or directive or already defined macro, the assembler tries to find the definition in a macro library directory. Multiple macro library directories may be used. The directory search order is defined by the `MACLIB` environment variable.

The assembler looks for the macro definition in a file whose name matches the operation field name in upper case or lower case with a lower case file extension of `.mac`. The `.mac` extension is required for a macro definition file on all platforms, unlike the `COPY` directive where any extension is supported and optional if the platform allows. Unless the platform's file system is case insensitive, for example, Windows, a file with a mixed case name will not be found.

When the macro definition file is found, the file is read defining each, or the single macro found within the file. Any statements other than comments or empty/blank lines are treated as an error, terminating the file's processing. Statements from the file are not included in the assembly listing. Only successfully defined macros become available to the assembler.

Macro Definition Mode

Macro definition mode is entered when the assembler directive `MACRO` is recognized during open code assembly. A user macro is defined within the constraints of the `MACRO` macro directive and a subsequent `MEND` macro directive. Later definitions of a macro with the same

name supersede previous definitions.

The first non-comment statement following the `MACRO` assembler directive must be the prototype statement.

Following the prototype statement is the optional macro body. The macro body consists of

- macro directives, described below, and
- model statements.

Macro directives are processed interpretively during the invocation of the macro. A macro is invoked when its name occurs in the operation field of an open code or macro model statement.

ASMA Specific Behavior: Errors occurring in macro directives or a macro's prototype statement will cause the macro definition to be ignored. Attempts to invoke a failed macro definition will result in an unrecognized statement.

ASMA Limitation: Nested macro definitions are not supported. A macro definition occurring within another macro definition will result in as many as six forms of errors.

1. The inner macro definition's `MACRO` statement will generate an error and will be ignored.
2. The inner macro definition's prototype statement will be treated as a model statement within the outer definition.
3. The inner macro's body statements will be erroneously treated as additional body statements of the outer macro definition.
4. The `MEND` associated with the inner macro definition will result in a premature end of the outer macro definition being interpreted as the `MEND` of the outer definition.
5. The outer macro's body statements following the MEND associated with the inner macro definition will be treated as occurring in open code.
6. The `MEND` associated with the outer macro definition, when finally encountered in open code, will generate an error.

Do not use inner macro definitions!

Macro Operation Identification

ASMA statement processing occurs in one of either two modes, driven by the operation identification processing within the current processing mode:

- assembly mode, see the “Assembly Operation Identification” section for details, or;
- macro definition mode.

During macro definition mode, only macro body statements participate in these steps. Statement positioning determines prototype statement recognition.

Step 1 – Identify a macro directive. Otherwise proceed to step 2.

Step 2 – Identify the statement as a model statement.

During macro invocation, assembly operation identification processing applies to the statements generated by a model statement, but not during macro definition mode.

Macro Invocation

A macro definition is invoked when it occurs in open code assembly or used in a model statement of a macro definition. The invocation creates a new input source. The statements of the macro definition in their internal form are interpreted. When a model statement is encountered it has any of its embedded symbolic parameters or variables replaced and is then submitted back to the assembler as an assembly statement.

ASMA Specific Behavior: Error handling during macro invocation uses a “fail-on-error” strategy. An error encountered during invocation of a macro directive causes immediate termination of the macro. The error is reported with the invoking assembler statement. Included in the error is the assembly statement of the failing macro definition.

Macro Symbols

Two types of symbols are unique to the macro language:

1. the variable symbol – any normal label name preceded by an ampersand, '`&`', and
2. the sequence symbol – any normal label name preceded by a period, '`.`'.

Sequence symbols may occur only in the name field of a macro definition's body. They are ignored in open code. The sequence symbol identifies where control is to be passed during the invocation of the macro. Sequence symbols are local to the macro.

Variable symbols arise in three contexts:

- prototype parameters, either as positional or keyword defined parameters
- global variable symbols or
- local variable symbols, and
- system variable symbols.

Within the context of an invoked macro, a single name space is used for all variable symbols. Symbol names occurring as parameters, globally declared or locally declared symbols must be unique.

Parameter variable symbols are “read-only” and are assigned values when the macro is invoked. All parameter variable symbols contain characters as if they were declared as local character variable symbols.

Global variable symbols are shared among macros. The first occurrence of the symbol's declaration creates it, after which it is shared between open code and invoked macros.

Local variable symbols are declared within the invoking macro and are only usable within the invoked macro. Undeclared local variable symbols are not supported.

System variable symbols will have a local or global context and type depending upon the symbol. System variable symbols all start with '`&SYS`'. User defined macros should not use these characters at the start of user defined variable symbols. Normal global and local symbol rules apply.

Variable symbols are of three types:

- arithmetic, assigned a signed numeric value, see the `LCLA` and `GBLA` macro directives, or
- binary, assigned a value of either zero or one, see the `LCLB` and `GBLB` macro directives, or
- character, assigned a string value, see the `LCLC` and `GBLC` macro directives.

In the following discussion, descriptions will use:

- `label` – for normal assembler location symbols,
- `var` – for a variable symbol and
- `seq` – for sequence symbols.

Subscripts

Globally or locally defined SET symbols may be defined with subscripts of one dimension. Subscripts require the use of a self defining value or an arithmetic expression to identify the subscript. Variable symbol subscripts must be in the range of one to the number of subscripts defined for the symbol. Implicit definitions can increase the size of a variable symbol.

Macro statement parameters and values associated with the `&SYSLIST` system variable symbol may be referenced by one or more subscripts, separated by commas and enclosed in parenthesis. Each subscript refers to the corresponding keyword or positional parameter or `&SYSLIST` sublist entry. A sublist may include its own sublists.

Variable Symbol Attributes

Three attributes are supported for macro symbolic variables, system variables and macro parameters:

- count (`K'`) – the count of characters used to represent the symbolic variable or referenced subscript.
- number (`N'`) – the number of subscripts defined for the symbolic variable or zero if not defined with subscripts.
- Type (`T'`) – symbol's type.

`K'` and `N'` attributes result in an arithmetic value that may be used anywhere a numeric value

is valid.

The count attribute always results in 0 for arithmetic or binary variable symbols regardless of the symbolic variable or subscripts actual value.

The number attribute is supported for all variable symbols regardless of their defined SET type.

System Variable Symbols

System variable symbols are read-only global or local symbols made available to a macro as if the corresponding global or local declaration had been made by the macro. The values established at the time a macro is invoked remain constant for the duration of that specific macro. The following system variable symbols are supported.

Symbol	Type	Description
`&SYSASM`	`GBLC`	Name of the assembler: `'A SMALL MAINFRAME ASSEMBLER'`
`&SYSCCW`	`LCLC`	`'CCWn'` directive used by the `CCW` directive, or `''` if not set
`&SYSCLOCK`	`LCLC`	Assembly UTC time in `'YYYY-MM-DD HH:MM:SS.mmmmmm'` format
`&SYSDATC`	`GBLC`	Assembly local date in `'YYYYMMDD'` format.
`&SYSDATE`	`GBLC`	Assembly local date in `'MM/DD/YY'` format
`&SYSDIR`	`GBLC`	Directory of the input file: `'/dir/path'` without `file.ext`
`&SYSECT`	`LCLC`	Name of the current active CSECT
`&SYSEXT`	`GBLC`	Input file's extension (may be empty): `'ext'` without the initial period
`&SYSFNAM`	`GBLC`	Input file name without an extension: `'file'` without `.ext`
`&SYSFNAME`	`GBLC`	Input file name with an extension: `'file.ext'`
`&SYSLIST`	`LCLC`	Macro statement positional operands as a single sublist
`&SYSLOC`	`LCLC`	Current location counter name, the same as `&SYSECT`.
`&SYSMAC`	`LCLC`	Macro name
`&SYSNDX`	`LCLC`	Macro index number
`&SYSNEST`	`LCLA`	Macro nesting level
`&SYSPATH`	`GBLC`	Full path of the input file: `'/dir/path/file.ext'`
`&SYSPSW`	`LCLC`	`'PSWxxx'` used by the `PSW` directive, or `''` if not set
`&SYSTIME`	`GBLC`	Assembly local time in `'HH.MM'` format
`&SYSVER`	`GBLC`	ASMA version in `'V.R.M'` format

Prototype Statement

```
[var]  <macname> [<var>[=<default>]][,<var>[=<default>]]...
```

Macro Definition Mode

The first statement in a macro definition following the `MACRO` assembler directive is the prototype statement.

The name field may contain an optional variable symbol.

The operation field contains the name of the macro. It follows the same rules as those of normal symbols. It is the name by which the assembler recognizes the macro when invoked during assembly.

The operand field contains a list of comma separated parameters. A positional parameter is a single variable symbol. A keyword parameter is a variable symbol, followed immediately by an '=' sign, followed immediately by a character sequence defining the keyword's default value.

Positional and keyword parameters may be mixed within the prototype statement.

Macro Invocation

If provided, the variable symbol in the name field will be assigned the character string value of the normal symbol occurring in the invoking statement's name field.

When the macro is invoked the positional parameter is assigned the string supplied for the corresponding positional parameter in the invoking statement. When the macro is invoked, keyword parameters take on the corresponding value assigned to the symbol by the invoking statement or its default value as defined in the prototype. Positional and keyword parameters are identified by the absence or presence, respectively, of a equal sign, '=', in the operand. Keyword parameters when invoked do not require the preceding ampersand for the variable symbol whose value is being specified.

Positional parameters are assigned values as they occur in the invoking statement. All positional parameters may be accessed via the `&SYSLIST` system variable symbol regardless of an explicit entry within the prototype for the position.

Model Statements

All statements within a macro definition that are neither a prototype statement nor a macro directive are model statements. Model statements are scanned for the occurrence of a variable symbol whose string value replaces the occurrence of the variable symbol. An optional period, '`.`', may be used to separate a variable symbol's name from any following statement characters that may follow.

If the variable symbol is defined with subscripts, a subscript must immediately follow the

variable symbol in the model statement. The subscript itself may refer to a defined SETA symbol.

Undefined variable symbols occurring in the a model statement will result in an error condition. Depending upon where the undefined variable symbol occurs, the resultant statement may trigger other assembler errors. Variable symbols occurring in open code are not replaced.

The resulting statement after variable replacement has occurred is submitted to the assembler language for processing and it is this resultant statement that appears in the listing, not the statement before substitution has occurred. Reference to the originating source statement in a macro definition is required to observe the original variable symbols before replacement.

Arithmetic Expressions

Arithmetic expressions support three types of operands:

- self-defining terms,
- previously defined symbolic variable references, and
- normal assembler labels.

Self defining terms are either a

- unsigned decimal number, any combination of number characters, '0' through '9'; a
- binary value of the form B'bb...', where each 'b' represents either a '0' or '1'; a
- hexadecimal value of the form X'hh...', where each 'h' represent a number '0' through '9' or upper or lower case 'a' through 'f'; or a
- character value of the form C'e' or CA'a' or CE'e'.

Character self defining terms allow for a single character. An EBCDIC character results from the translation of the ASCII source character after conversion by the current code page assembler setting.

An arithmetic expression supports four dyadic operations: addition ('+'), subtraction ('−'), multiplication ('*') and division ('/'). Division discards any encountered remainder.

Two monadic operations are supported: positive ('+') and negative ('−'). A signed self defining term is treated as having a monadic operator.

Parenthesis have the usual effect of overriding operator precedence.

References to assembler labels must result in an integer as a value, either through calculation or by assignment within the assembly.

ASMA Limitation: Arithmetic expressions may not contain any literal specifications.

Logical Expressions

Logical expressions support two types of operands:

- arithmetic expressions including self-defining terms as described in the “Arithmetic Expressions” section;
- symbolic variable references previously defined by either a `GBLA`, `GBLB`, `LCLA` or `LCLB` directive, with or without subscripts as dictated by the symbol definition; or
- character strings, with or without symbolic replacement as occurring for model statements.

A character string is two single quotation marks with zero or more intervening characters, other than another single quotation mark. Sub-string specifications including a starting character position, relative to one, and a length are optional.

Arithmetic symbolic variables as operands reflect the value as compared to zero. An arithmetic value equal to zero is treated as a binary value of zero. Otherwise the value is considered to be a binary one in the logical expression.

An arithmetic symbolic variable may be compared to a self-defining term or another arithmetic symbolic variable. A character string may be compared to another character string. When character strings are compared, their EBCDIC character coding values are used as established by the current active code page translation. See the “Code Pages” section for details on using built-in code page definitions or providing a user defined code page definition.

The following comparisons are supported. Comparison operators compare value to the left of the operator to the value to the right of the operator. Comparison operates must be in upper case.

- `EQ` – the left hand value equal comparison
- `NE` – not equal comparison,
- `LT` – less than comparison,
- `LE` – less than or equal comparison,
- `GT` – greater than comparison, and
- `GE` – greater than or equal comparison.

Logical expression operands or the results of comparison operators may be combined with logical relationships. Logical relationships combine the operand or comparison operator result on the left with the operand or comparison operator on the right. The following relationships are supported and must be in upper case.

- `AND`,
- `AND NOT`,
- `OR`,

- `OR NOT`,
- `XOR`, and
- `XOR NOT`.

A logical relationship, operand or comparison result may be reversed using the monadic `NOT` operator, in upper case.

The precedence of these operations are from highest to lowest, the highest being performed first if not explicitly controlled via expression parentheses:

- `NOT` monadic operator
- comparison operators
- relationship operators.

The result of a logical expression is either a binary zero (interpreted as False) or a binary one (interpreted as True).

ASMA Limitation: Logical expressions may not contain any literal specifications.

Macro Directives

Macro directives may have roles during either macro definition mode or during macro invocation or both.

ACTR

```
[seqsym]  ACTR  <arithmetic-expression>
```

The `ACTR` directive modifies the current setting of macro local branch counter value. The ACTR value is decremented each time an `AGO` or `AIF` directive is executed.

Macro Definition Mode

The arithmetic expression is required.

Macro Invocation

The arithmetic expression is evaluated. The result updates the current setting of the macro local branch counter variable, ACTR. Negative values are treated as zero.

AGO

Unconditional Branch Syntax:

```
[seqsym] AGO <seq>
```

Computed Branch Syntax:

```
[seqsym] AGO <(arithmetic-expression)><seq>[,seq]...
```

The AGO directive causes either an unconditional branch within a macro to the statement identified by the required sequence symbol (the unconditional branch syntax) or a branch conditional on the results of the required arithmetic expression to one or more locations (the computed branch syntax). The AGO directive itself may contain a sequence symbol in its name field. In both cases as least one sequence symbol is required. In the case of the computed branch syntax, additional sequence symbols may be supplied, each separated by a comma.

Macro Definition Mode

In both cases, the AGO directive is converted to an internal representation. Sequence symbols are validated as defined within the macro at the completion of the macro definition.

The arithmetic expression in the computed branch syntax must be enclosed within a parenthesis pair and be immediately followed by the initial required sequence symbol. Processing of the arithmetic expression follows that of the SETA directive.

Macro Invocation

The unconditional branch syntax form of the directive causes an immediate transfer of control to the macro body statement to which the sequence symbol is associated.

The computed branch syntax form of the directive, first calculates the result of the arithmetic expression. If the result is between one and number of sequence symbols provided, control is transferred to the statement associated with the corresponding sequence symbol. If the result of the arithmetic expression falls outside of this range, control flows to the next sequential statement immediately following the AGO directive.

See the “Arithmetic Expressions” section for details related to macro language arithmetic expressions.

Each time the AGO directive is executed the macro local branch counter value, ACTR, is decremented. If the ACTR value is zero when the directive is encountered, the current macro invocation is aborted with an error. Unless altered by an ACTR directive, the branch counter value is initially set to 4096.

AIF

```
[seqsym] AIF (<logical-expression>)<seqsym>,...
```

The AIF directive performs a conditional transfer of control within a macro definition. Each operand requires a logical expression, enclosed in a parenthesis pair, immediately followed by a sequence symbol defined in the name field of a macro directive or model statement. The alternative statement format is supported for the AIF directive. At least one operand is required.

Macro Definition Mode

The logical expression and sequence symbol of each operand is converted to an internal form for later evaluation. See the “Logical Expressions” sections for details.

Macro Invocation

The logical expression of each operand is evaluated in turn. The first expression resulting in a binary one (implying True) passes control to the statement defining the referenced sequence symbol and operand logical expression evaluation ends. Otherwise, the logical expression of the next operand is evaluated. If none of the operands evaluate to a binary one, control passes to the next sequential macro statement.

Each evaluated operand decrements the local branch counter value, ACTR. If the ACTR value is zero when the evaluation occurs, the current macro invocation is aborted with an error. Unless altered by an ACTR directive, the branch counter value is initially set to 4096.

ANOP

```
[seqsym] ANOP [comment]
```

Macro Definition Mode

The ANOP directive provides a placeholder for defining a sequence symbol, if provided.

Macro Invocation

The ANOP performs no operation during macro invocation.

GBLA

```
[seqsym] GBLA <sym[(sub)]>[,sym[(sub)]...
```

Macro Definition Mode

The GBLA directive defines one or more global arithmetic symbolic variables. The leading '&' is defined symbolic variable is optional.

If the optional subscript is provided for the symbol, the symbol will be defined with subscripts and may only be referenced with a subscript. The subscript must be a decimal self-defining term.

A sequence symbol in the name field is optional.

Each defined symbol is converted to an internal form used during macro invocation.

Macro Invocation

Each symbol identified in the `GBLA` directive is defined in the global macro symbol table if not already defined and initialized with a value of zero.

If the symbol is already defined its original definition is used.

The global variable is then made available to the macro with its current value if it does not conflict with a prototype parameter or another symbol of the same name already available to the macro.

GBLB

```
[seqsym] GBLB <sym[(sub)]>[,sym[(sub)]...
```

Macro Definition Mode

The `GBLB` directive defines one or more global binary symbolic variables. The leading '`&`' is defined symbolic variable is optional.

If the optional subscript is provided for the symbol, the symbol will be defined with subscripts of one dimension and may only be referenced with a subscript. The subscript must be a decimal self-defining term.

A sequence symbol in the name field is optional.

Each defined symbol is converted to an internal form used during macro invocation.

Macro Invocation

Each symbol identified in the `GBLB` directive is defined in the global macro symbol table if not already defined and initialized with a value of zero implying false.

If the symbol is already defined its original definition is used.

The global variable is then made available to the macro with its current value if it does not conflict with a prototype parameter or another symbol of the same name already available to the macro.

GBLC

```
[seqsym] GBLC <sym[(sub)]>[,sym[(sub)]...
```

Macro Definition Mode

The GBLC directive defines one or more global character symbolic variables. The leading '&' is defined symbolic variable is optional.

If the optional subscript is provided for the symbol, the symbol will be defined with subscripts of one dimension and may only be referenced with a subscript. The subscript must be a decimal self-defining term.

A sequence symbol in the name field is optional.

Each defined symbol is converted to an internal form used during macro invocation.

Macro Invocation

Each symbol identified in the GBLC directive is defined in the global macro symbol table if not already defined and initialized with a value of the empty character string (''').

If the symbol is already defined its original definition is used.

The global variable is then made available to the macro with its current value if it does not conflict with a prototype parameter or another symbol of the same name already available to the macro.

LCLA

```
[seqsym] LCLA <sym[(sub)]>[,sym[(sub)]...
```

Macro Definition Mode

The LCLA directive defines one or more local arithmetic symbolic variables. The leading '&' is defined symbolic variable is optional.

If the optional subscript is provided for the symbol, the symbol will be defined with subscripts of one dimension and may only be referenced with a subscript. The subscript must be a decimal self-defining term.

A sequence symbol in the name field is optional.

Each defined symbol is converted to an internal form used during macro invocation.

Macro Invocation

Each symbol identified in the LCLA directive is defined in the local macro symbol table, initialized with a value of zero, and made available to the macro provided the symbol does not conflict with a prototype parameter or another symbol of the same name already made available to the macro.

LCLB

```
[seqsym] LCLB <sym[(sub)]>[,sym[(sub)]...
```

Macro Definition Mode

The `LCLB` directive defines one or more local binary symbolic variables. The leading '`&`' is defined symbolic variable is optional.

If the optional subscript is provided for the symbol, the symbol will be defined with subscripts of one dimension and may only be referenced with a subscript. The subscript must be a decimal self-defining term.

A sequence symbol in the name field is optional.

Each defined symbol is converted to an internal form used during macro invocation.

Macro Invocation

Each symbol identified in the `LCLB` directive is defined in the local macro symbol table, initialized with a value of zero, implying false, and made available to the macro provided the symbol does not conflict with a prototype parameter or another symbol of the same name already made available to the macro.

LCLC

```
[seqsym] LCLC <sym[(sub)]>[,sym[(sub)]...
```

Macro Definition Mode

The `LCLC` directive defines one or more local character symbolic variables. The leading '`&`' is defined symbolic variable is optional.

If the optional subscript is provided for the symbol, the symbol will be defined with subscripts of one dimension and may only be referenced with a subscript. The subscript must be a decimal self-defining term.

A sequence symbol in the name field is optional.

Each defined symbol is converted to an internal form used during macro invocation.

Macro Invocation

Each symbol identified in the `LCLC` directive is defined in the local macro symbol table, initialized with a value of the empty string (`''`), and made available to the macro provided the symbol does not conflict with a prototype parameter or another symbol of the same name already made available to the macro.

MACRO

```
MACRO [DEBUG] [comment]
```

Causes the assembler to exit open code assembly and enter macro definition mode. Any optional label is ignored. If the optional `DEBUG` operand is provided, macro definition debug output will be generated. Otherwise, any text provided in the operand field is treated as a comment. The macro directive `MEND` causes the assembler to leave macro definition mode and return to open code assembly. See the “Macro Language” section for details on usage of the `MACRO` and `MEND` directives.

The `MACRO` directive must not provide any label field.

MEND

```
[seqsym] MEND [comment]
```

Macro Definition Mode

Terminates and completes a macro definition. A completed macro definition contains an internal representation of the definition. `MEND` causes the assembler to leave macro definition mode and return to open code assembly mode. An optional sequence symbol, if provided, is incorporated as part of the defined macro. Any text provided in the operand field is treated as a comment.

If errors were encountered during macro definition mode, the macro definition fails and it is not created. This case is reported as an error occurring during `MEND` processing. Subsequent statements attempting to utilize the failed macro definition will themselves result in unrecognized operations.

Macro Invocation

When interpreted during macro invocation, the `MEND` statement terminates the invocation. Macro interpretation ceases and the macro ceases to be an input source to open code assembly.

MEXIT

```
[seqsym] MEXIT [comment]
```

Macro Definition Mode

An optional sequence symbol as allowed. Any content of the statement operand field is treated as a comment.

Macro Invocation

Causes the macro to pass control to the macro's `MEND` directive.

SETA

```
<var[(subscript)]> SETA <arithmetic expression>[,...]
```

Macro Definition Mode

The name field of the statement must contain a variable symbol which may optionally contain a subscript enclosed in parenthesis. If provided the subscript may be a self defining term or an arithmetic expression. It identifies to what symbol or its subscript the result of the arithmetic expression will be be assigned.

The arithmetic expression is recognized and converted into an internal representation used during macro invocation. Lexical errors are recognized during macro definition mode. Some syntactical errors will not be detected until macro invocation.

Macro Invocation

The arithmetic expression is evaluated. The current value of the subscript (if provided) is determined. The result is assigned to the `GBLA` or `LCLA` variable symbol or its specified subscript. When the symbol is subscripted, each additional operand's arithmetic expression is evaluated and assigned to the next sequential subscript. If the symbol being set has not previously been declared, it will be implicitly declared as a local symbolic variable. It's value will then be set as described.

The presence or absence of a subscript must be consistent with the definition of the symbolic variable or an error is recognized. Symbolic variables defined with a subscript must have a subscript when setting its value. Symbolic variables not defined with a subscript must not have a subscript when setting its value

Certain syntactical errors may be detected during invocation. Errors related to symbol definitions and subscript values (for example out of range for the symbols definition) are detected during invocation.

See the “Arithmetic Expressions” section for details related to macro language arithmetic expressions.

SETB

```
<var[(subscript)]> SETB <logical expression>[,...]
```

Macro Definition Mode

The name field of the statement must contain a variable symbol which may optionally contain a subscript enclosed in parenthesis. If provided the subscript may be a self defining term or an arithmetic expression. It identifies to what symbol or its subscript the result of the logical expression will be be assigned. At least one logical expression is required.

The logical expression is recognized and converted into an internal representation used during macro invocation. Lexical errors are recognized during macro definition mode. Some syntactical errors will not be detected until macro invocation.

Macro Invocation

The logical expression is evaluated. The current value of the subscript (if provided) is determined. The result is assigned to the `GBLB` or `LCLB` variable symbol or its specified subscript. When the symbol is subscripted, each additional operand's logical expression is evaluated and assigned to the next sequential subscript.

If the symbol being set has not previously been declared, it will be implicitly declared as a local symbolic variable. It's value or values will then be set as described.

The presence or absence of a subscript must be consistent with the definition of the symbolic variable or an error is recognized. Symbolic variables defined with a subscript must have a subscript when setting its value. Symbolic variables not defined with a subscript must not have a subscript when setting its value.

Certain syntactical errors may be detected during invocation. Errors related to symbol definitions and subscript values (for example out of range for the symbols definition) are detected during invocation.

See the “Logical Expressions” section for details related to macro language logical expressions.

SETC

```
<var[(subscript)]> SETC <operand>[,...]
```

Operands may be a character expression:

```
'characters'[(start,length)]
```

an attribute of a macro parameter, symbolic variable or system variable:

```
X'&VAR[(subscript,..)]
```

or an attribute of an assembler label:

```
X'LABEL
```

Macro Definition Mode

The name field of the statement must contain a variable symbol which may optionally contain a subscript enclosed in parenthesis. If provided the subscript may be a self defining term or an arithmetic expressionl. It identifies to what symbol or its subscript the result of the character expression(s) will be be assigned.

The character expression, symbol attribute, or label attribute is recognized and converted into an internal representation used during macro invocation. Lexical errors are recognized during macro definition mode. At least one operand is required. Some syntactical errors will not be detected until macro invocation.

Macro Invocation

The character expression is evaluated, the symbol attribute or label attribute is determined. The current value of the subscript (if provided) is determined. The result is assigned to the variable symbol or its specified subscript. When the symbol is subscripted, each additional operand is evaluated and assigned to the next sequential subscript.

If the symbol being set has not previously been declared, it will be implicitly declared as a local symbolic variable. It's value or values will then be set as described.

Character expression evaluation occurs in three steps. The second and third occur only if the optional sub-string specification is provided.

1. Perform symbolic variable substitution on the supplied characters between the two single quotation marks. The string may be empty. The same rules used by model statements for symbolic variable symbol replacement applies to the `characters` within the `SETA` operand. See the “Model Statements” section above.
2. If the sub-string specification is provided, calculate the starting position in the string by evaluating the `start` arithmetic expression and calculate the length of the sub-string by evaluating the `length` arithmetic expression. See the section “Arithmetic Expressions” on details of how the two values are specified. The `start` value is relative to 1 and any value less than 1 or greater than the string expression length results in an error. The `length` value includes the starting position. The `start` position plus the `length` must not result in a value exceeding the string expression length.
3. Extract the sub-string from the character sequence. A `length` value of zero results in an empty string regardless of the valid `start` position.

Label attributes are derived from the currently defined label. If undefined, the attribute default value is provided, except for the case of the `D'` attribute, which provides a numeric value depending upon the current state of the assembler label's definition. Numeric attributes are converted into decimal self-defining terms.

Symbolic attributes are derived from the current values for the symbolic variable. Numeric

attributes are converted into decimal self-defining terms. Omitted macro parameters or sublist entries are defined as empty strings with a `T'` attribute of `O`. In the case of a `SETC` symbol or macro parameter, if the value is a valid assembler label, the label's attribute is provided.

The result is then assigned to the previously defined `GBLC` or `LCLC` or implicitly defined symbolic variable. Errors relating to the sub-string or subscript `start` and `length` values are detected during macro invocation.

ASMA Specific Behavior: `SETC` symbols, or macro parameters, within expressions do not have to be embedded within quoted strings for character comparisons. For example, the `&CHARS` symbol can be used like this:

```
     &STATE  SETB  &CHARS EQ 'ABC'
```

In cases where compatibility with other assemblers is important, all string references to a `SETC` symbol or macro parameter must be in a quoted string, like this:

```
     &STATE  SETB  '&CHARS' EQ 'ABC'
```

Failure to include the single quotation marks will result in an error from such assemblers.

ASMA supports both coding styles.

Appendix A - Instruction Source Formats

Each mainframe instruction falls into a specific format. All instructions that share the same format also share the same assembler source syntax and binary machine instruction structure. Binary machine instructions identify their components by a letter and number.

Operands in assembler source are separated by commas. All operands are required unless otherwise indicated by a Note. The source syntax or operands are identified in the following table by the heading columns with “Op1” through “Op5”. The operand description includes an alphabetic component and a number. The alphabetic designation indicates the kind of operand and the numbered portion of the machine instruction format.

Assembler syntax operand ordering does not always match instruction operand. For example, in the RIE-d format, the machine instruction identifies the immediate field as its second operand. However, the assembler syntax places the immediate field as its third operand.

The actual formats and instructions supported by ASMA is controlled by the contents of an external database coded using the Machine Specification Language. The command line argument identifying the targeted machine dictates instruction support. The following tables are informational.

Syntax Summary by Instruction Format

Format	Example	Length	Opcode	Op1	Op2	Op3	Op4	Op 5
E bits	PR	2	0-15	-	-	-	-	
I bits	SVC	2	0-7	I1 8-15	-	-	-	-
IE bits	NIAI	4	0-15	I1 24-27	I2 28-31	-	-	-
MII bits	BPRP	6	0-7	M1 8-11	RI2 12-23	RI3 24-47	-	-
RI a bits	IIHL	4	0-7, 12-15	R1 8-11	I2 - Note 1 16-31	-	-	-
RI b bits	BRAS	4	0-7, 12-15	R1 8-11	RI2 16-31	-	-	-
RI c bits	BRC	4	0-7, 12-15	M1 – Note 1 8-11	RI2 16-31	-	-	-
RIE a bits	CGIT	6	0-7 40-47	R1 8-11	I2 16-31	M3 32-35	-	-
RIE b	CGRJ	6	0-7,	R1	R2	M3	RI4	-

Format	Example	Length	Opcode	Op1	Op2	Op3	Op4	Op 5
bits			40-47	8-11	12-15	32-35	16-31	
RIE c bits	CGIJ	6	0-7, 40-47	R1 8-11	I2 32-39	M3 12-15	RI4 16-31	-
RIE d bits	AHIK	6	0-7, 40-47	R1 8-11	R3 12-15	I2 16-31	-	-
RIE e bits	BRXHG	6	0-7, 40-47	R1 8-11	R3 12-15	RI2 16-31	-	-
RIE f bits	RXSBG	6	0-7, 40-47	R1 8-11	R2 12-15	I3 16-23	I4 24-31	I5 32-29
RIL a bits	LGFI	6	0-7. 12-15	R1 8-11	I2 16-47	-	-	-
RIL b bits	LARL	6	0-7 12-15	R1 8-11	RI2 16-47	-	-	-
RIL c bits	BRCL	6	0-7, 12-15	M1 8-11	RI2 16-47	-	-	-
RIS bits	CGIB	6	0-7, 40-47	R1 8-11	I2 32-29	M3 12-15	D4(B4) B4 16-19 D4 20-31	-
RR bits	BCR BALR	2	0-7	M1/R1 8-11	R2 – Note 1 12-15	-	-	-
RRD bits	MAEBR	4	0-15	R1 16-19	R3 24-27	R2 28-31	-	-
RRE bits	LPEBR	4	0-15	R1 - Note 1 24-27	R2 – Note 1 28-31	-	-	-
RRF a bits	MDTR	4	0-15	R1 24-27	R2 28-31	R3 16-19	M4 – Note 1 20-23	-
RRF b bits	QADTR	4	0-15	R1 24-27	R3 16-19	R2 28-31	M4 – Note 1 20-23	-
RRF c bits	CGRT	4	0-15	R1 24-27	R2 28-31	M3 – Note 1	-	-
RRF d bits	LDETR	4	0-15	R1 24-27	R2 28-31	M4 20-23	-	-
RRF e bits	CELFBR	4	0-15	R1 24-27	M3 16-19	R2 28-31	M4 – Note 1 20-23	-
RRS bits	CGRB	6	0-7, 40-47	R1 8-11	R2 12-15	M3 32-35	D4(B4) B4 16-19 D4 20-31	-
RS a bits	BXH	4	0-7	R1 8-11	R3 – Note 1 12-15	D2(B2) B2 16-19 D2 20-31	-	-
RS b bits	STCM	4	0-7	R1 8-11	M3 12-15	D2(B2) B2 16-19	-	-

Format	Example	Length	Opcode	Op1	Op2	Op3	Op4	Op 5
						D2 20-31		
RSI bits	BRXH	4	0-7	R1 8-11	R3 12-15	RI2 16-31	-	-
RSL a bits	TP	6	0-7 40-47	D1(L1,B1) L1 8-11 B1 16-19 D1 20-31	-	-	-	-
RSL b bits	CDZT	6	0-7, 40-47	R1 32-36	D2(L2,B2) L2 8-15 B2 16-15 D2 20-31	M3 36-39	-	-
RSY a bits	LMY	6	0-7, 40-47	R1 8-11	R3 12-15	D2(B2) B2 16-19 DL2 20-31 DH2 32-29	-	-
RSY b bits	CLMH	6	0-7, 40-47	R1 8-11	M3 12-15	D2(B2) B2 16-19 DL2 20-31 DH2 32-40	-	-
RX a bits	STH	4	0-7	R1 8-11	D2(X2,B2) X2 12-15 B2 16-19 D2 20-31	-	-	-
RX b bits	BC	4	0-7	M1 8-11	D2(X2,B2) X2 12-15 B2 16-19 D2 20-31	-	-	-
RXE bits	LDEB	6	0-7, 40-47	R1 8-11	D2(X2,B2) X2 12-11 B2 16-19 D2 20-31	-	-	-
RXF bits	MAEB	6	0-7, 40-47	R1 32-35	R3 8-11	D2(X2,B2) X2 12-11 B2 16-19 D2 20-31	-	-
RXY a bits	LRAG	6	0-7, 40-47	R1 8-11	D2(X2,B2) X2 12-15 B2 16-19 DL2 20-31 DH2 32-39	-	-	-
RXY b bits	PFD	6	0-7, 40-47	M1 8-11	D2(X2,B2) X2 12-15 B2 16-19 DL2 20-31 DH2 32-39	-	-	-
S	STIDP	4		D2(B2)	-	-	-	-

Format	Example	Length	Opcode	Op1	Op2	Op3	Op4	Op 5
bits			0-15	B2 16-19 D2 20-31				
SI bits	TM	4	0-8	D1(B1) B1 16-19 D1 20-31	I2 8-15	-	-	-
SIL bits	MVHHI	6	0-15	D1(B1) B1 16-19 D1 20-31	I2 32-47	-	-	-
SIY bits	TMY	6	0-7, 40-47	D1(B1) B1 16-19 DL1 20-31 DH1 32-39	I2	-	-	-
SMI bits	BPP	6	0-7	M1 9-11	RI2 32-47	D3(B3) B2 16-19 D3 20-31	-	-
SS a bits	TRTR	6	0-7	D1(L,B1) L 8-15 B1 16-19 D1 20-31	D2(B2) B2 32-35 D2 36-47	-	-	-
SS b bits	MVO	6	0-7	D1(L1,B1) L1 8-11 B1 16-19 D1 20-31	D2(L2,B2) L2 12-15 B2 32-35 D2 36-47	-	-	-
SS c bits	SRP	6	0-7	D1(L1,B1) L1 8-11 B1 16-19 D1 20-31	D2(B2) B2 32-35 D2 36-47	I3 12-15	-	-
SS d bits	MVCK	6	0-7	D1(R1,B1) R1 8-11 B1 16-19 D1 20-31	D2(B2) B2 32-35 D2 36-47	R3 12-15	-	-
SS e bits	PLO	6	0-7	R1 8-11	D2(B2) B2 16-19 D2 20-31	R3 12-15	D4(B4) B4 32-35 D4 36-47	
SS f bits	PKA	6	0-7	D1(B1) B1 16-19 D1 20-31	D2(L2,B2) L2 8-15 B2 32-35 D2 36-47	-	-	-
SSE bits	LASP	6	0-15	D1(B1) B1 16-19 D1 20-31	D2(B2) B2 32-35 D2 36-47			
SSF bits	MVCOS	6	0-7, 12-15	D1(B1) B1 16-19 D1 20-31	D2(B2) B2 32-35 D2 36-47	R3 8-11		

Note 1: Not defined for all instructions of this format.

Extended Mnemonics

Extended mnemonics for relative instructions come with two forms, one using `BRxxx` for BRANCH RELATIVE ON CONDITION modeled on the extended mnemonics for BRANCH ON CONDITION. The other form uses `Jxxxx` to imply a relative instruction. ASMA will only provide the Jxxxx form of extended mnemonics for relative instructions.

Mnem.	Actual	Implied	Op1	Op2	Op3	Op4	Op5
B	BC	M1=15	D2(X2,B2)				
BR	BCR	M1=15	R2				
NOP	BC	M1=0	D2(X2,B2)				
NOPR	BCR	M1=0	R2				
BH	BC	M1=2	D2(X2,B2)				
BHR	BCR	M1=2	R2				
BL	BC	M1=4	D2(X2,B2)				
BLR	BCR	M1=4	R2				
BE	BC	M1=8	D2(X2,B2)				
BER	BCR	M1=8	R2				
BNH	BC	M1=13	D2(X2,B2)				
BNHR	BCR	M1=13	R2				
BNL	BC	M1=11	D2(X2,B2)				
BNLR	BCR	M1=11	R2				
BNE	BC	M1=7	D2(X2,B2)				
BNER	BCR	M1=7	R2				
BP	BC	M1=2	D2(X2,B2)				
BPR	BCR	M1=2	R2				
BM	BC	M1=4	D2(X2,B2)				
BMR	BCR	M1=4	R2				
BZ	BC	M1=8	D2(X2,B2)				
BZR	BCR	M1=8	R2				
BO	BC	M1=1	D2(X2,B2)				
BOR	BCR	M1=1	R2				
BNP	BC	M1=13	D2(X2,B2)				
BNPR	BCR	M1=13	R2				
BNM	BC	M1=11	D2(X2,B2)				
BNMR	BCR	M1=11	R2				

Mnem.	Actual	Implied	Op1	Op2	Op3	Op4	Op5
BNZ	BC	M1=7	D2(X2,B2)				
BNZR	BCR	M1=7	R2				
BNO	BC	M1=7	D2(X2,B2)				
BNOR	BCR	M1=7	R2				
J	BRC	M1=15	RI2				
JLU	BRCL	M1=15	RI2				
JNOP	BRC	M1=0	RI2				
JLNOP	BRCL	M1=0	RI2				
JH	BRC	M1=2	RI2				
JLH	BRCL	M1=2	RI2				
JL	BRC	M1=4	RI2				
JLL	BRCL	M1=4	RI2				
JE	BRC	M1=8	RI2				
JLE	BRCL	M1=8	RI2				
JNH	BRC	M1=13	RI2				
JLNH	BRCL	M1=13	RI2				
JNL	BRC	M1=11	RI2				
JLNL	BRCL	M1=11	RI2				
JNE	BRC	M1=7	RI2				
JLNE	BRCL	M1=7	RI2				
JP	BRC	M1=2	RI2				
JLP	BRCL	M1=2	RI2				
JM	BRC	M1=4	RI2				
JLM	BRCL	M1=4	RI2				
JZ	BRC	M1=8	RI2				
JLZ	BRCL	M1=8	RI2				
JO	BRC	M1=1	RI2				
JLO	BRCL	M1=1	RI2				
JNP	BRC	M1=13	RI2				
JLNP	BRCL	M1=13	RI2				
JNM	BRC	M1=11	RI2				
JLNM	BRCL	M1=11	RI2				
JNZ	BRC	M1=7	RI2				
JLNZ	BRCL	M1=7	RI2				
JNO	BRC	M1=14	RI2				

Mnem.	Actual	Implied	Op1	Op2	Op3	Op4	Op5
JLNO	BRCL	M1=14	RI2				
JAS	BRAS	none	R1	RI2			
JASL	BRASL	none	R1	RI2			
JCT	BRCT	none	R1	RI2			
JCTG	BRCTG	none	R1	RI2			
JXH	BRXH	none	R1	R3	RI2		
JXHG	BRXHG	none	R1	R3	RI2		
JXLE	BRXLE	none	R1	R3	RI2		
JXLEG	BRXLG	none	R1	R2	RI2		

Appendix B - Use of Machine Specification Language (MSL)

Machine Specification Language (MSL) is a small domain specific language used to define instructions and system architectures. It is a subset of the Statement Oriented Parameter Language (SOPL). SOPL is implemented as a **Python** module and is imported by the **Python** module implementing any language based upon it, MSL included. ASMA's MSL files are contained in the SATK `asma/msl` directory.

The specific instructions supported by a given architecture are defined by the architecture's MSL file. MSL defines for ASMA an instruction's assembler source format and its machine code bit representation. The machine code bit representations are defined by a set of formats found in the `asma/msl/formats.msl` file. All instruction formats are contained within this file. Each architecture specific file includes the `formats.msl` file regardless of whether all formats are used by the architecture or not.

Historically, system *Principles of Operations* manuals have defined machine instruction formats in a manner *similar* to the formats used in MSL. This creates a set of expectations for formats within MSL. While inspired by such historical formats, MSL formats are not the same. They are driven by the needs of ASMA, not a hardware CPU. The primary purpose of this section is to describe these differences. Inevitably some discussion of how ASMA operates in this context is necessary, sometimes diving into **Python**.

Instruction Formats

Instruction formats in the historical context define how specific bits of an instruction are used by the hardware decoding instructions. The 16, 32, or 48 bits of the instruction are broken up into multiple bit fields of varying length. Each field with its defined role in the operation of the instruction is identified. Some fields are 4 bits. Some fields are 12 bits. Some are even 32 bits in length. While the role is defined, the impact on the instruction is articulated in the individual instruction's description including how the field's bits are constructed.

This consideration is illustrated by the case of two register-immediate instructions: `LHI` and `TML`. Legacy formats use the same format for these two instructions: `RIA`. ASMA has a format with this name. The issue is how the immediate value is built. Legacy instruction formats do not care.

However, the MSL instruction format, also named `RIA`, treats the immediate field as signed. The format, as defined by MSL is below. The critical line is highlighted in **bold**.

This format works just fine for `LHI`.

```
#     0     4     8     12    16    20    24    28   31
#     +-----+-----+-----+-----+-----+-----+-----+-----+
#     |    OP     | R1  | XOP |          RI2          |
#     +-----+-----+-----+-----+-----+-----+-----+-----+
#
# RIA: MNEMONIC R1,RI2
#
format RIA
    length 4
    xopcode 12 15
    mach R1 8 11
    mach RI2 16 31 signed
    source R1 R1
    source RI2 RI2
```

However, this does not work so well for `TML`. `TML` treats the four-byte immediate field as an *unsigned* value. Because MSL requires all elements of the format to apply to an instruction, an entirely different format is required, `RIAU`, in this case.

```
#     0     4     8     12    16    20    24    28   31
#     +-----+-----+-----+-----+-----+-----+-----+-----+
#     |    OP     | R1  | XOP |          RI2          |
#     +-----+-----+-----+-----+-----+-----+-----+-----+
#
# RIAU: MNEMONIC R1,RI2
#
format RIAU
    length 4
    xopcode 12 15
    mach R1 8 11
    mach RI2 16 31
    source R1 R1
    source RI2 RI2
```

The only change is in the single `mach RI2` parameter. The parameter value of `signed` has been removed, turning the field into an unsigned value.

Why does it matter? ASMA uses the parameter to determine if a value defined within an assembly will fit into the field as an arithmetic value. Using, for example, `X'FFFFFFFF'`, for the value when the instruction's format is `RIA` results in an assembly error.

So, while one instruction format suffices in the legacy world, ASMA requires two.

MSL instruction formats also define the sequence of assembly source operands using the `source` parameter. If the sequence of source formats are different, even though the instruction format is the same, a different MSL format is required. There is therefore a many-to-one relationship between MSL instruction formats and legacy formats.

Consequently, the MSL format names may be different. MSL tries to maintain some similarity to legacy format names, but this is not strictly maintained.

Statement Names

All MSL statements require a name. All MSL statement names must be unique. This applies to MSL instruction formats. One needs to differentiate between MSL statements and paramaters. MSL at its heart is an instance of a Statement Oriented Parameter Language. SOPL statements must have names.

While MSL was being designed it was viewed as a system configuration vehicle. A number of statements were implemented with that in mind. However, as time progressed, it became evident that for the foreseeable future, only ASMA would utilize MSL. As a consequence, the module that processes an MSL file, has been placed within ASMA as `asma/msldb.py`. The `msldb.py` module reads the MSL file and creates an object `msldb.MSLDB`. This is essentially a **Python** dictionary. **Python** dictionaries allow a key-to-object relationship. In the case of MSL, the key is the statement's name. Each object within the `MSLDB` object is an instance of `msldb.MSLDBE` and corresponds to a statement within the MSL file.

CPUX Objects

The `MSLDBE` entries are not ideal for use by the assembler. These still maintain a separation of CPU and instruction information including the formats. To assist the assembler the CPU object within MSLDB provides a method, `expand`. This method combines all of the CPU information into one object, a `msldb.CPUX` object.

The actual CPU and MSL file that drives an assembly is part of the parameters supplied to ASMA itself, either explicitly or implicitly. While explicit parameters are possible, generally the assembler `--target` argument is used to specify the instruction set architecture targeted for the assembly. Refer to the "-t/--target ISA" section of "Assembler Controls" for details on this mapping. The `CPUX` object drives many aspects of the assembler, not just supported instructions and their formats.

This object has proven to be so successful that it is used by the `tools/mslrpt.py`

command line tool for report generation of instruction support and most other reports. Whenever an MSL file changes instruction support, usually to add instructions, the `tools/mslrpt.py` tool is used to produce the text file `asma/msl/PoO.txt`. The `PoO.txt` file identifies each instruction that is supported by each architecture in a table format similar to that found in *Principles of Operations* manuals. The instructions are provided in three formats: instruction mnemonic, instruction machine operation code, and instruction format. The `CPUX` object makes this and other reports possible.