Table of Contents

Introduction

Machine Specification Language (MSL) is a small domain specific language describing key elements of a mainframe system. MSL's primary role is support for consumers of mainframe configuration information. To the extent a consumer's needs can be addressed within the broad design of MSL, those needs will be addressed. MSL serves the consumers, not the other way around. The following constitutes the current description of the MSL.

It is the responsibility of users of a file written in MSL to parse and process it as appropriate to its needs. It is hoped that the language is simple enough that other scripting languages, for example REXX, or Perl, etc., can do that. A Python processor and two command line utilities are provided. The command line utility is intended for testing purposes and validation of a file written in MSL. A file successfully processed by the command line utility should be considered as valid.

Language Overview

The Machine Specification Language is a line oriented language. It does not support free flowing text. A 'line' does NOT include platform specific line termination characters. A process needs to handle removal of such characters and tolerate variations. If a processor is not able to do that, an external mechanism should convert the lines to the platform specific termination before being processed.

A valid line contains only the ASCII characters 0x20-0x7E, inclusive. Comments are excluded from this requirement.

Lines fall into one of these descriptions:

1. an “empty” line (contains only spaces or no characters at all).
2. a “comment” line (starting with either a '*' or '#' in position 1),
3. an “invalid” line (contains characters other than ASCII 0x20-0x7E inclusive),
4. a “statement” line (printable character(s) starting in position 1), or
5. a “parameter” line (printable character(s) starting in position 2 or later),

Only statement and parameter lines are processed. A processor is also free to ignore any statement or parameter line for which it has no interest. Statement and parameter lines may also contain a comment starting with a comment character through the remainder of the line. Character restrictions do not apply to comments.

An '`id`' identifies each statement. Statement id's must be unique. Duplicate identifiers are not allowed. All identifiers reside in a single name space.

The MSL language is a specific use of the underlying Statement Oriented Parameter Language (SOPL). SOPL is implemented in the `sopl.py` module in the `tools/lang` directory. SOPL was designed with the express objective that tools based on languages other than Python could be developed if applicable without resorting to complex language processing tools.

Case Sensitivity

MSL is generally **case sensitive**. Case sensitivity is the default. All statement types, parameter types and parameter or statement attributes not explicitly identified as being case insensitive are case sensitive.

For convenience certain identifiers or attributes are treated as case insensitive resulting in such being treated as upper case:

- `inst` statement identifiers,
- `format` statement identifiers,
- `format` statement `mfield` attributes, and
- `format` statement `sfield` attributes.

Statement Consistency

Taken together, the individual statements define a hierarchical database describing instructions supported by a CPU. The CPU is at the top of the hierarchy. The other statements are subsidiary to the CPU:

```
cpu <cpu-id>
     iset <is-id>
          inst <mnemonic>
               format <format-id>
```

Individual statements receive an identifier, identified above between the <> symbols. Identifiers are placed in a single name space that forces all identifiers to be unique. Identifiers from an included MSL file are placed in this same name space and must not clash with identifiers from other files.

Instruction mnemonics specified by the collection of `iset` statements referenced by a CPU must be disjoint. While a single mnemonic may be defined in more than one `iset`, the mnemonic may only occur once in all of the `iset` statements used by a single CPU.

An assembler uses the information supplied by an `inst` statement defining the instruction and its referenced `format` statement combined with the assembly supplied information to construct the machine binary representation of an instruction. The format statement defines both the structure and source of each instruction field used to build the instruction. The inst source parameter lines define how the an assembly source operand is used to supply specific binary fields. Some fields may be implied by the instruction mnemonic, for example some extended mnemonics, and the `inst` statement must include a `fixed` parameter line supplying that information. The corresponding `mach` parameter defining the field must include the attribute `fixed` in these cases.

`format` Statement Requirements

If there are differences in the machine structure or source statement operands, separate `format` statements are required. Extended mnemonics require a different `format` statement than that used in the defining base instruction's `inst` statement. Some formal instruction format definitions contained in *Principles of Operation* manuals contain fields that are not used by some instructions utilizing the formal format. MSL requires separate formats in this case.

Many extended mnemonics specify a value for a field within the second byte of the instruction. MSL can utilize the `inst` statement `opcode` definition, with the appropriately defined `format` statement `XOP` `mach` definition to supply the extended mnemonic mask value. For extended mnemonics where the mask is not within the second byte of the instruction, the `inst` statement `fixed` parameter can supply the value (with the corresponding `fixed` attribute associated with the field's `mach` parameter line).

Python Integration

Integration of the MSL processor into another Python module is accomplished by importing the `msldb` module. The importing module will instantiate the `MSL` object and call its external methods to drive its use of the processor. The `MSL` object's `DB()` method delivers a completed and validated database. The database is primarily accessed like a dictionary using a string as its index. The string corresponds to a statement's identification.

The `msl.py` module in the `tools` directory provides a command line interface to the MSL processor and serves as an example of how the database is integrated for use by a separate module. The `mslrpt.py` module in the `tools` directory reports on MSL database content. Details on each tool can be found in the “Command-Line Utilities” section.

MSL Statements

include

```
include path
```

Includes another MSL file into the current file. **Spaces** within the path are **not** supported.

cpu

```
cpu <cpu-id> [exp]
    addrmax <16|24|31|64>
    ccw <CCW0|CCW1>
    psw <PSWS|PSW360|PSW67|PSWBC|PSWEC|PSW380|PSWXA|PSWE370|
         PSWE390|PSWZ>
    base <iset-id> ... <iset-id>
    features <iset-id> ... <iset_id>
    exclude <mnemonic> ... <mnemonic>
    include <mnemonic> ... <mnemonic>
```

The '`cpu`' statement defines a specific Central Processing Unit. The '`cpu`' statement is focused on defining the instructions supported by the processor. The optional '`exp`' statement attribute indicates the CPU is experimental.

Standard instruction sets are defined by one or more '`base`' parameter lines. Optional instruction sets are identified by one or more '`features`' parameter lines. Each instruction set is defined by its own '`iset`' statement. Some processors support additional instructions not part of a feature. Such additional instructions are defined by their mnemonic in one or more '`include`' parameter lines. Correspondingly some processors do not support the full set of instructions in some instruction sets. Identification of such unsupported instructions is handled by identifying the instruction mnemonic of the unsupported instruction in one or more '`exclude`' parameter lines.

The '`addrmax`', '`ccw`' and '`psw`' parameter lines define aspects related to creating CPU specific structures. The '`addrmax`' parameter line defines the maximum valid address for the CPU in number of bits. The only valid attribute values for '`addrmax`' are 16, 24, 31, and 64. The '`ccw`' parameter line identifies the CCW format expected to be used by the CPU. The only valid attributes for the '`ccw`' parameter line are `CCW0` and `CCW1`. Upper case is required. The '`psw`' parameter line accepts a number of formats for its attribute. Upper case is required. See the available options in the '`psw`' model parameter line above. These three parameter lines are provided primarily for the use of an assembler.

An example of CPU would be '`2025`' identifying a System/360 model 25.

iset

```
iset <is-id>
   mnemonics <mnemonic> ... <mnemonic>
```

The '`iset`' statement defines an instruction set. It is composed of one or more '`mnemonics`' parameter lines identifying the instructions that are part of the set.

inst

```
inst <mnemonic> <opcode> <format-id> [flags]
    [fixed <mfield> <value>]
    [filter <mfield> <name>]
```

The '`inst`' statement defines a single instruction. All of its attributes are specified as part of the '`inst`' statement itself. One or more optional `fixed` parameters are supported by the `inst` statement.

The '`inst`' statement uses an instruction's assembler mnemonic as its identification. As with all of the statements, the identification immediately follows the '`inst`' statement type.

Following the mnemonic is the instruction's hexadecimal operation code. The code may be between two and four hexadecimal digits depending upon whether an extended code is used.

Next is the machine instruction format definition. It is the identification of a '`format`' statement that defines the layout of the instruction in memory and its assembler source operand sequence.

The final attribute is optional. It sets certain flags. The flags are coded together as a single attribute, that is not separated by any spaces. All flags must be coded in upper case. The following flags are recognized by the '`inst`' statement:

- 'P' – If present, the instruction is privileged and if omitted the instruction is not privileged
- 'E' – If present, the instruction is an extended mnemonic recognized by an assembler only.
- 'X' – If present, the instruction is experimental.
- 'L' – if present, the `format` statement `length` parameter overrides the implied length of bits 0 and 1 of the `inst` statement `opcode` parameter. If omitted, the `format` statement `length` parameter and bits 0 and 1 of the `inst` statement `opcode` parameter must be consistent.

An example of an instruction statement is this:

```
inst AR 1A RR
```

The `fixed` parameter always references an instruction field by its `<mfield>` name as

defined in the instruction's `format` statement. `fixed` is used to specify a hexadecimal value as content of the machine instruction field.

`filter` identifies a filter to be applied to the fields value as coded for the instruction. The following field filters are supported by ASMA instruction builder module, `insnbldr.py`.

- NOP – No operation,
- `TAONE` – Treat bit zero as a 1, or
- `TAZERO` – Treat bit zero as a 0.

Filters are applied immediately prior to inserting the field's value into the instruction.

The `fixed` parameter is primarily used when defining an extended mnemonic that provides an implicit value for a field (the `number` attribute). The `filter` parameter is used when an extended mnemonic operand is modified by the assembler when building the instruction.

Parameters `fixed` and `filter` are mutually exclusive.

format

```
format <format-id>
    length <bytes>
    xopcode <start-bit> <end-bit>
    mach <mfield> <start-bit> <ending-bit> [signed] [fixed]
    source <sfield> <mfield> <mfield>...
```

A '`format`' statement defines:

- the layout of an instruction in storage (`mach` parameter lines),
- the syntax of the corresponding assembly language coding of the instruction (`source` parameter lines), and
- the source for each instruction field.

The identification contained in the statement line is referenced by '`inst`' statements.

Four types of parameter lines are recognized: '`length`', '`xopcode`', '`mach`', and '`source`'. The only optional parameter is the '`xopcode`' parameter line. One and only one '`length`' parameter line is required. Each field within the layout of the instruction requires a '`mach`' parameter line. Each operand of the an assembly source statement requires a '`source`' parameter line.

The '`length`' parameter line defines the length of the instruction in memory. Only three values are accepted: 2, 4, and 6.

The '`mach`' parameter line identifies by a name, the '`mfield`' attribute and its starting and ending bit positions within the machine instruction. Bit 0 is always the first bit of the instruction and bit positions 15, 31 or 47 are the last bit of the instruction depending upon the

length. The last attribute of the 'mach' parameter line indicates if the field content is a signed or unsigned value. The presence of the string 'signed' as the fourth or fifth attribute indicates a signed value. If the signed attribute is omitted, it indicates the value is unsigned. See the section “Requirements for 'mfield' and 'sfield' Attributes”. The presence of the string 'fixed' as the fourth or fifth attribute indicates the source of the field is supplied by a 'fixed' parameter in a referencing inst statement. The source parameter is omitted for mfield’s supplied by the inst statement by means of the fixed parameter.

Special handling of a mach field by an assembler may be specified by reserved mfield names. For such reserved mfield names, the assembler must recognize the name and the special handling implied by it. Currently the only reserved mach mfield name is:

- RXB – special handling for vector instruction assembler supplied value.

A 'source' parameter line describes the syntactical requirement of an assembler when assembling an instruction. A 'source' parameter line is required for each assembly language operand. This parameter identifies to which machine instruction fields the operand contributes values. All of the identified machine fields must have been specified in its own 'mach' parameter line. 'source' parameter lines may be placed in any location relative to the other parameter lines. However, the sequence of the source parameter lines themselves determines the sequence of the individual comma separated operands in an assembler statement. See the section “Requirements for 'mfield' and 'sfield' Attributes”.

The optional 'xopcode' locates within the instruction where the extended operation code value is placed within the instruction. Bits 0-7 of all instructions contain the primary operation code and is automatically defined. This parameter identifies where the additional 4 or 8 bits of the instruction are place.

An example of a format statement showing the separate identification of two register fields and corresponding assembler operands.

```
format RR
    length 2
    mach R1 8 11
    mach R2 12 15
    source R1 R1
    source R2 R2
```

Requirements for 'mfield' and 'sfield' Attributes

The 'mfield' and 'sfield' attributes have the appearance of being identifiers like those used in statement lines. Identifiers have few restrictions. However, these two attributes communicate type information and identification of a machine field or source operand when more than one such field or operands occurs in the machine instruction or source operands.

In general these attributes use the following structure:

```
XX...Xnn...n
```

The 'x' indicates an alphabetic character, specifically any upper case letter between 'A' and 'Z', inclusive, of which the attribute may include one or more letters. The 'n' indicates one or more numeric characters, specifically any character between '0' and '9', inclusive.

The preceding description is the general structure of these attributes. The character portion of the attribute actually identifies the type of machine instruction field or source statement operand is being defined. The numeric portion of the attribute is simply an identification allowing multiple such fields or operands to be differentiated within the '`format`' statement. As specification of a type each attribute is restricted to specific character sequences.

The following table describe the recognized '`mfield`' types. In each case a number can be associated with a specific type to differentiate it from another machine instruction field of the same type.

`mfield` Type	**Description**
`B`	A base register field
`D`	A displacement field
`DH`	The high portion of a displacement field
`DL`	The low portion of a displacement field
`I`	An immediate field
`M`	A mask field
`R`	A register field
`RELI`	An instruction relative immediate field
`RI`	An instruction immediate field
`RXB`	Vector register number overflow field
`V`	A vector register field
`X`	An index register field

The following table describes the assembler source '`sfield`' types. Unlike the machine field, the source operand describes the syntax required by an assembler for a given operand. In the syntax fields of the following table, the strings '`exp`', '`len`', '`ndx`' and '`reg`' refer to an arithmetic expression. In each case a number can be associated with a specific type to differentiate it from another source operand of the same type. An example would be `R1` and `R2` to differentiate two `R` type `mfield`'s or `sfield`'s in one instruction.

`sfield` Type	**Assembler Supported Operand Syntax**	
`I`	`exp`	
`M`	`exp`	

sfield Type	**Assembler Supported Operand Syntax**			
R	exp			
V	exp			
RELI	exp			
RI	exp			
S or SY	exp	exp(exp)		
SR	exp	exp(reg)	exp(,exp)	exp(reg,exp)
SL or SYL	exp	exp(len)	exp(,exp)	exp(len,exp)
SX or SYX	exp	exp(ndx)	exp(,exp)	exp(ndx,exp)

The primary difference between the 'sfield' types that contain a 'Y' and the corresponding type without the 'Y' relates strictly to needs of the machine instructions with which they are used. In each of these cases the operand syntax required by an assembler is the same. However the needs of the corresponding machine instructions are different with regards to their treatment of displacement fields. Likewise, the 'RELI' type imposes specific requirements on an assembler for instruction relative immediate machine fields.

Command-Line Utilities

Two utilities are available for use of MSL:

- `msl.py` – Validates MSL source file content
- `mslrpt.py` – MSL database reporting tool

The foundation for both tools is the `msldb.py` module that processes and creates the internal representation of an MSL file.

System Requirements

Both tools require 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 the MSL tools. The tools are dependent upon the tools provided by the Stand-alone Tool Kit, SATK. SATK is available from github:

https://github.com/s390guy/SATK

The following directories are dynamically added to the Python directory search order.

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

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

Normally use of the `PYTHONPATH` environment variable is not required.

Environment Variables

One search path environment variable consistent with the platform's conventions is supported:

- `MSLPATH` – defines the search path for MSL `include` statements.

If not provided the default directory `${SATK_DIR}/asma/msl` is used to locate included files. The `MSLPATH` environment variable is not normally required.

By convention MSL source files use the `.msl` suffix, but are not required to do so.

msl.py – MSL Validation

The `msl.py` tool validates an MSL file and all of its included files. It is primarily useful during MSL file development. The tool validates the basic coding requirements and internal consistency of the information within the files. Only use of the information in the files can validate the correctness of the information within the files. A file successfully processed by the `msl.py` utility should be considered as valid.

The command line syntax is as follows:

```
[python[3.3]] msl.py [script-file-argument]... mslfile
```

The `[python[3.3]]` command-line argument is environment dependent. Some environments may require it and some may not. Refer to your platform's Python documentation whether it is required.

`msl.py` identifies the tool being invoked and is required.

Following the tool name one or more script file arguments may be supplied.

The final required `mslfile` argument identifies the MSL file being validated. This argument may be either an absolute path or relative path. If a relative path or file name alone is provided, the `MSLPATH` environment variable or default directory is used to locate the file.

Script File Arguments

The following optional arguments are supported.

-e/--expand

If specified, expand the identified `cpu` statement's definition into its complete set of instructions and supported formats. The identified `cpu` statement must be defined in the validated MSL file.

-f/--fail

If specified the validation terminates on the first failure encountered.

-x/--xref

If specified a cross-reference listing is produced.

--dump

If specified the internal Python objects representing the database are displayed. This is primarily used during MSL processor debugging.

mslrpt.py – MSL Reporting

The `mslrpt.py` tool is useful in producing information related to the overall MSL database environment and file content.

The command line syntax is as follows:

```
[python[3.3]] mslrpt.py [script-file-argument]...
```

The `[python[3.3]]` command-line argument is environment dependent. Some environments may require it and some may not. Refer to your platform's Python

documentation whether it is required.

`mslrpt.py` identifies the tool being invoked and is required.

Following the tool name is the one required and optional script-file arguments.

Script File Arguments

The `-r` or `--report` is the only required script file argument. It identifies the type of report to be generated. All other arguments are optional depending upon the selected report and if not associated with the selected report ignored. Regardless of whether an option is used by a report or not, if supplied it must be recognized by the tool.

-r/--report cpu

The `cpu` report argument causes all defined CPU's and their defining file names within the `MSLPATH` to be identified. This is useful in determining valid options for other tools that may rely upon the value for its processing.

-r/--report files

The `files` report argument causes all MSL files, identified by their `.msl` suffix to be identified in the order in which they occur based upon the `MSLPATH` definition or files identified in the default directory.

-r/--report inst

The `inst` report argument causes the identified `cpu` definitions' instruction sets and related statistics to be reported. The report builds a table of instructions incorporating the defined instructions for each selected CPU. The instructions are then reported based upon their mnemonics, operation codes, or format in sequence. Optionally the report may be written to a file or displayed to the user. The former option is recommended for large reports.

For each instruction available to the selected CPU's, its mnemonic, operation code and format are included. For each selected CPU a column is built in the instruction table containing the instructions status for that CPU.

The following status information is presented:

'`-`' indicates the instruction is not available on this CPU,

'`E`' the instruction is recognized as an assembler extended mnemonic,

'`G`' indicates the CPU supports the instruction as a non-privileged,

'`L`' indicates the instruction operation code does not conform the the normal instruction length implied by the operation code's bit 0 and 1 values (very rare),

'`P`' indicates the CPU supports the instruction as a privileged instruction, and

'X' the instruction is supported as experimental by this CPU.

More than one status indicator may be present for a CPU.

-c/--cpu mslfile=cpu

The `--cpu` argument identifies a `cpu` definition and its MSL file. The identified CPU's instructions are included in the report. The `mslfile` portion of the argument's value identifies the MSL file. If the `.msl` extension is required it must be included. The file is located by means of the `MSLPATH` environment variable or default directory. The `cpu` portion of the argument's value identifies the `cpu` statement within the MSL file whose instructions are included in the report. Multiple `--cpu` arguments may be provided, adding each CPU to the report.

The `files` and `cpu` reports are intended to facilitate CPU identification.

-l/--listing filepath

This optional argument identifies the file to which the report is written. If omitted, the report is written to `sysout`. If the `filepath` argument value is a relative path, the current working directory is used. Otherwise the `filepath` argument value must be an absolute path.

-s/--seq <mnemonic|opcode|format>

The `--seq` argument specifies sequence in which the instructions will appear in the report. One of the following values is required:

`mnemonic` indicating the instructions are sequenced by instruction mnemonic

`opcode` indicating the instructions are sequenced by instruction operations code value

`format` indicating the instructions are sequenced by instruction format.

The `--seq` argument may be included more than once. Each unique setting will cause the instructions to be sequenced using the specified value. As many as three separate instruction lists may be generated by a report.

-r/--report PoO

The `PoO` report is a special case of the `inst` report. A preselected set of files and CPU definitions are used matching the architectural definitions offered in the various *Principles of Operation* manuals. The `--listing` and `--seq` arguments apply to the `PoO` report with the same meanings as for the `inst` report. If specified, the `--cpu` argument is ignored. See the information in the "`-r, --report inst`" section for argument descriptions.

The `--report` argument value `PoO` is case sensitive.