High Level Assembler Plugin

User documentation

Michal Bali, Marcel Hruška, Peter Polák,
Adam Šmelko, Lucia Tódová

Supervisor: Miroslav Kratochvíl

Consultant: Slavomír Kučera (Broadcom)

Contents

Introduction

HLASM Language Support is an extension for Visual Studio Code that adds support for the High Level Assembler. It provides code completion, highlighting and navigation features, detects common mistakes in the source, and lets you trace the evaluation of the conditional assembly source code, using a modern debugging interface.

In this document we describe how to use the extension. The document is organized as follows:

- Section 1 covers the installation of the plugin and its configuration for basic usage
- Section 2 quickly reviews all basic features of the extension from the user perspective
- Section 3 contains a description of configuration necessary for using macro libraries and COPY members

1 Getting Started

1.1 Installation

The extension requires a working installation of Visual Studio Code[1] to work.

There are two possibilities for installation:

Installation from VS Marketplace The extension can be installed from the Visual Studio Code Marketplace[2] by following these steps:

[1] `https://code.visualstudio.com/`

[2] `https://marketplace.visualstudio.com/items?itemName=broadcomMFD.hlasm-language-support`

1. Open the extensions tab (Ctrl + Shift + X)
2. Search for “HLASM language support” and select it.
3. Select “Install”.

Installation from source The extension can be built on Windows, Mac OS and most Linux distributions using the standard CMake build procedure. A detailed guide for building and installing the extension from source can be found in the Programmer documentation, chapter 10.

1.2 Usage

Opening a HLASM project is done as such:

1. In **File** - **Open Folder...**, select the folder with the HLASM sources. (An example workspace is provided in the folder `example_workspace`.)
2. Open any HLASM source file (note that HLASM does not have a standard filename extension) or create a new file.
3. If the auto-detection of HLASM language does not recognize the file, set it manually in the bottom-right corner of the VS Code window.
4. The extension is now enabled on the opened file. If you have macro definitions in separate files or use the COPY instruction, you need to setup the workspace.

1.3 Setting up a multi-file project environment

HLASM COPY instruction copies the source code from various external files, as driven by HLASM evaluation. The source code interpreter in the HLASM Extension needs to be set up correctly to be able to find the same files as the HLASM assembler program.

This is done by setting up two configuration files — `proc_grps.json` and `pgm_conf.json`. The extension guides the user in their creation:

1. After opening a HLASM file for the first time, two pop-ups are displayed. Select `Create pgm_conf.json with current program` and `Create empty proc_grps.json`. The two configuration files are then created with default values. They are written into the `.hlasmplugin` subfolder.
2. Navigate to the `proc_grps.json` file. This is the entry point where you can specify paths to macro definitions and COPY files. To do this, simply fill the `libs` array with the corresponding paths. For example, if you have your macro files in the `ASMMAC/` folder, add the string `"ASMMAC"` into the libs array.

Follow section 3 for more detailed instructions for configuring the environment.

2 Functionality of extension

The HLASM Language Support extension parses and analyzes all parts of a HLASM program. It resolves all ordinary symbols, variable symbols and checks the validity of most

instructions. The extension supports conditional and unconditional branching and can define global and local variable symbols. It can also expand macros and COPY instructions.

2.1 Highlighting

The HLASM Language Support extension highlights statements with different colors for labels, instructions, operands, remarks and variables. Statements containing instructions that can have operands are highlighted differently to statements that do not expect operands. Code that is skipped by branching AIF, AGO or conditional assembly is not colored (see fig. 1).

2.2 Autocomplete

Autocomplete is enabled for the instruction field. While typing, a list of instructions starting with the typed characters displays. Selecting an instruction from the list completes it and inserts the default operands. Variables and sequence symbols are also filled with a value from their scope (see fig. 2).

2.3 Go To Definition and Find All References

The extension adds the functionality of `go to definition` and `find all references`. Use the `go to definition` functionality to show definitions of variable symbols, ordinary symbols and macros, or open COPY files directly. Use the `find all references` functionality to show all places where a symbol is used fig. 3).

2.4 Macro Tracer

The macro tracer functionality allows you to track the process of assembling HLASM code. It lets you see step-by-step how macros are expanded and displays values of variable symbols at different points during the assembly process. You can also set breakpoints in problematic sections of your conditional assembly code.

The macro tracer is not a debugger. It cannot debug running executables, only track the compilation process.

2.4.1 Configuring the Macro Tracer

1. Open your workspace.
2. In the left sidebar, click the bug icon to open the debugging panel (Ctrl + Shift + D).
3. Click create a launch.json file. A `select environment` prompt displays.
4. Enter **HLASM Macro tracer**. Your workspace is now configured for macro tracing.

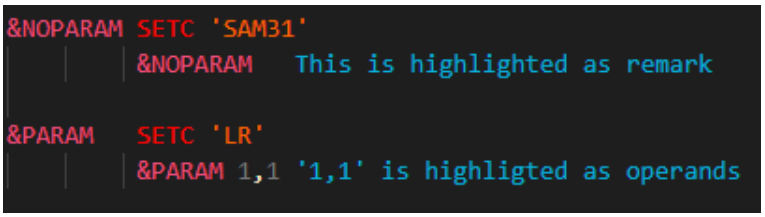

Figure 1: HLASM code with syntax highlighting.

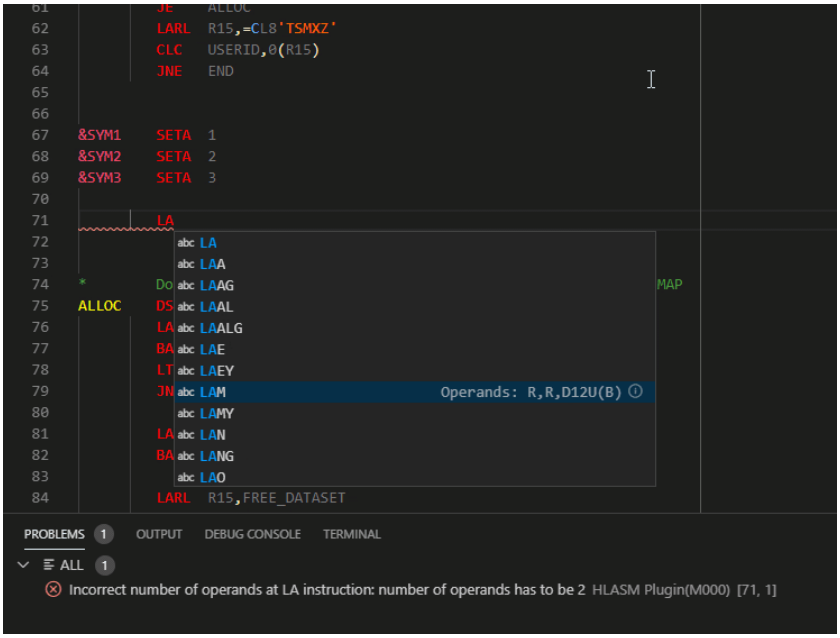

Figure 2: Autocomplete usage example.

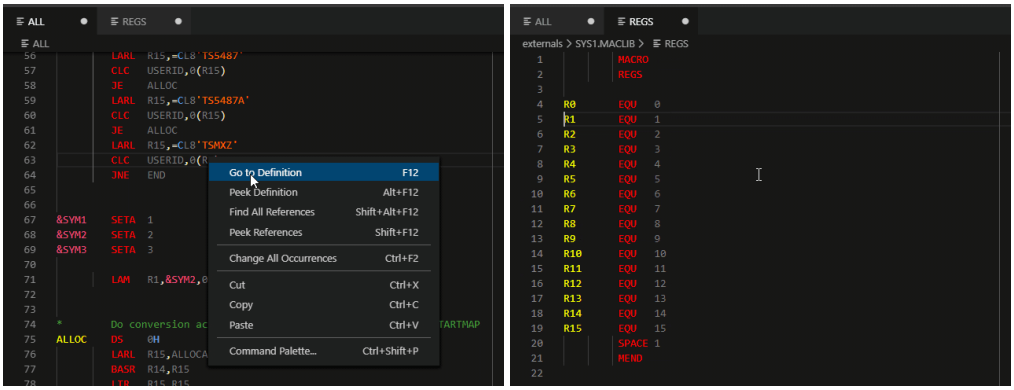

Figure 3: Go To Definition usage example; the editor opens the place of definition of the selected R1 symbol.

2.4.2 Using the Macro Tracer

To run the macro tracer, open the file that you want to trace. Then press **F5** to open the debugging panel and start the debugging session.

When the tracer stops at a macro or COPY instruction, you can select **step into** to open the macro or COPY file, or **step over** to skip to the next line.

Breakpoints can be set before or during the debugging session (see fig. 4).

3 External Macro Libraries and COPY Members

The HLASM Language Support extension looks for locally stored members when a macro or COPY instruction is evaluated. The paths of these members are specified in two configuration files in the `.hlasmplugin` folder of the currently open workspace:

- `proc_grps.json` defines **processor groups** by assigning a group name to a list of directories. Hence, the group name serves as a unique identifier of a set of HLASM libraries defined by a list of directories.
- `pgm_conf.json` provides a mapping between **programs** (open-code files) and processor groups. It specifies which list of directories is used with which source file. If a relative source file path is specified, it is relative to the current workspace.

Therefore, to use a predefined set of macro and copy members, do the following steps:

1. Enumerate the library directories in `proc_grps.json` and name them with an identifier; thus, create a new processor group.
2. Use the identifier of the new processor group with the name of your source code file in `pgm_conf.json` to assign the library members to the program.

The structure of the configuration is based on CA Endevor® SCM.

An example configuration can be seen in listing 1 and listing 2. It describes two processor groups `GROUP1`, `GROUP2` and their respective mappings to programs `source` and `asm_prg`. If you have the two example configuration files and invoke the `MAC1` macro from `source`, the folder `ASMMAC/` in the current workspace is searched for a file named exactly `MAC1`. If that file is not found, the folder `C:/SYS.ASMMAC` is searched. If that search is unsuccessful too, an error displays that the macro does not exist.

Note that the macro `MAC1` is searched in directories in order as they are listed in the configuration.

To add a possibility to specify a custom extension for HLASM files in the workspace, pgm_conf.json has an optional field `alwaysRecognize`. It takes an array of wildcards and provides the following functionality:

- All files matching these wildcards will always be recognized as HLASM files. Hence, the HLASM plugin will be automatically running on all the matched files.
- If an extension wildcard is defined, all macro and copy files with such extension may be used in the source code. For example, with the extension wildcard `*.hlasm`, a user may add macro MAC to his source code even if it is in a file called `Mac.hlasm`.

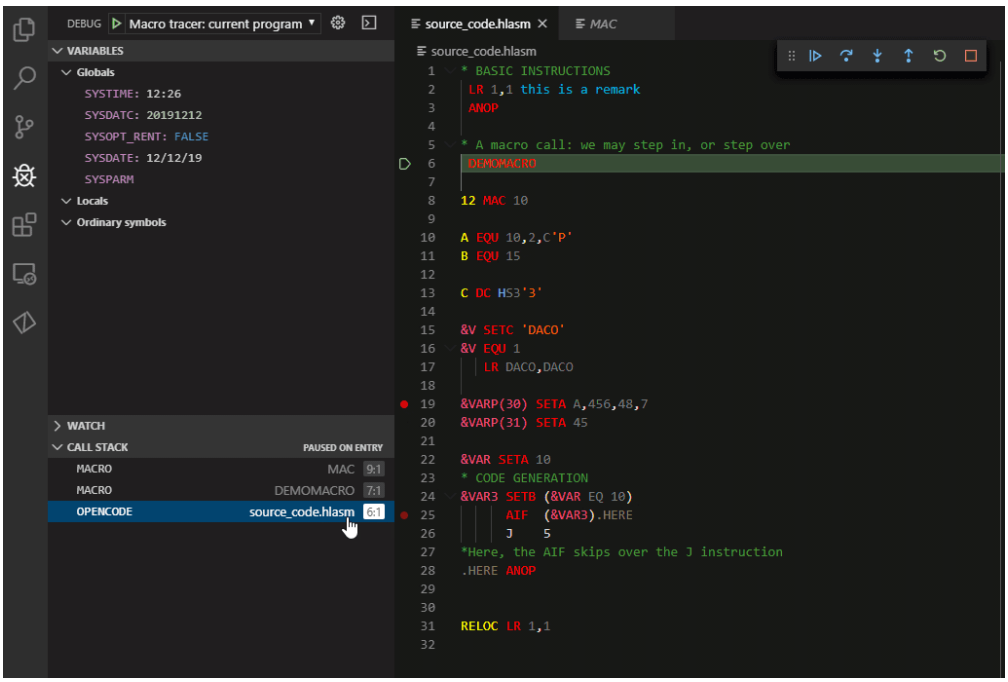

Figure 4: Macro tracer interface, showing HLASM source code with highlighted current instruction, call stack and variable listing.

Listing 1 An example of `proc_grps.json` configuration file with two processor groups.

```
{
  "pgroups": [
    {
      "name":"GROUP1",
      "libs": [
        "ASMMAC/",
        "C:/SYS.ASMMAC"
      ]
    },
    {
      "name":"GROUP2",
      "libs": [
        "G2MAC/",
        "C:/SYS.ASMMAC"
      ]
    }
  ]
}
```

Listing 2 An example of `pgm_conf.json` configuration file with two programs.

```
{
  "pgms": [
    {
      "program": "source",
      "pgroup": "GROUP1"
    },
    {
      "program": "asm_prg",
      "pgroup": "GROUP2"
    },
  ]
}
```

Listing 3 A `pgm_conf.json` example with the `alwaysRecognize` field.

```
{
  "pgms": [
  {
    "program": "source",
    "pgroup": "GROUP1"
  }
  ],
   "alwaysRecognize" : ["*.hlasm", "libs/*.asm"]
}
```

Listing 4 An example of a program wildcard.

```
{
  "pgms": [
  {
    "program": "*",
    "pgroup": "GROUP1"
  }
  ]
}
```

Listing 3 demonstrates the mentioned functionality. With this configuration file, processor group `GROUP1` will be assigned to `source` and `source.hlasm` file as well. Also, macro and copy files in the `lib` directory can be referenced and correctly recognized in the program without the `.asm` extension.

Finally, the program field in `pgm_conf.json` supports wildcards as well (see listing 4). Using this feature, users can easily share one processor group among all their programs.