

Hopper SDK Quick Tour

It is now easy to extend the capabilities of Hopper. Thanks to its SDK, you can write your own support for a given file format, or even a specific CPU. This document will give you an overview of the whole architecture.

SDK Setup

In order to start developing plugins, you'll need to download the SDK archive from the official website (<https://www.hopperapp.com/download.html>).

Mac

On macOS, you don't need any special setup before using the SDK. You can double-click on the `Samples/samples.xcworkspace` file to explore the samples projects.

⚠ WARNING: Starting with macOS 11.0, the plugins need to be properly signed with your Apple developer certificate in order to be loaded by Hopper.

Linux

On Linux, you'll need to compile your own version of **GNUstep**. Hopper makes use of this framework for its core features, in order to mimic the Mac behaviour. First, make sure that you have all the necessary tools. On Ubuntu, you can install all of them using this command:

```
% sudo apt-get install build-essential git subversion autoconf automake cmake \
  libffi-dev libxml2-dev libgnutls-dev libicu-dev libblocksruntime-dev \
  libkqueue-dev libpthread-workqueue-dev autoconf libtool clang curl
```

Once done, create a directory, unarchive the SDK, and execute the script under **Linux/install.sh**.

```
% mkdir Hopper_dev
% cd Hopper_dev
% unzip ~/Downloads/HopperSDK-4.x.x.zip
% cd Linux
% ./install.sh
```

The script takes a few minutes to complete. Once done, you'll find a new directory **gnustep-Linux-x86_64**. GNUstep requires some environment variables in order to compile projects properly. To set up your environment, you can source the GNUstep script:

```
% source ./gnustep-Linux-x86_64/share/GNUstep/Makefiles/GNUstep.sh
```

Now you can compile the sample projects.

```
% cd ../Samples/Linux
% make
```

The bundles should have been compiled in the current directory. You should create a symbolic link to the final destination, so that Hopper will find them:

```
% mkdir -p ~/GNUstep/Library/ApplicationSupport/Hopper/PlugIns/v4/{CPUs,Loaders,Tools}
% ln -s $(pwd)/AmigaLoader.bundle \
~/GNUstep/Library/ApplicationSupport/Hopper/PlugIns/v4/Loaders/AmigaLoader.hopperLoader
% ln -s $(pwd)/M68kCPU.bundle \
~/GNUstep/Library/ApplicationSupport/Hopper/PlugIns/v4/CPUs/M68kCPU.hopperCPU
% ln -s $(pwd)/SampleTool.bundle \
~/GNUstep/Library/ApplicationSupport/Hopper/PlugIns/v4/Tools/SampleTool.hopperTool
```


Plugins

Hopper supports three types of plug-in: **File Loader Plugin**, **CPU Plugin**, and **Tool Plugin**. In each case, the plugin is a simple *Cocoa Bundle* which declares a class that conforms to the `HopperPlugin` Objective-C protocol. In the SDK archive, you'll find a sample projects for each of them.

The bundle contains a `Info.plist` file, which references a main class, using the `NSPrincipalClass` key. This is the class to be instantiated by Hopper.

The plugin can be installed at the system or the user scope. The base paths are:

- For the system scope: `/Library/Application Support/Hopper/PlugIns/v4`
- For the user scope: `~/Library/Application Support/Hopper/PlugIns/v4`

 This is not a typo: the `v4` part in the path refers to the internal architecture, not the version of the Hopper application.

Depending on its type, a plugin needs to be installed in a subdirectory, following this rule:

- For a *File Loader Plugin*, Hopper searches in a `Loaders` subdirectory for all bundles with the `hopperLoader` file extension,
- For a *CPU Plugin*, Hopper searches in a `CPU` subdirectory for all bundles with the `hopperCPU` file extension,
- For a *Tool Plugin*, Hopper searches in a `Tools` subdirectory for all bundles with the `hopperTool` file extension.

For instance, the Motorola 68000 CPU plugin is installed in

```
~/Library/Application Support/Hopper/PlugIns/v4/CPUs/M68kCPU.hopperCPU.
```

The Common Interface

Every plugin must conform to the `HopperPlugin` Objective-C protocol.

This protocol gives information to Hopper on the type of plugin that is implemented, and various textual information about the plugin's name, the author, etc.

When Hopper starts, it loads, and instantiates each plugin it finds in the different directories. A plugin is initialised by Hopper using the `-[HopperPlugin initWithHopperServices:]` selector. An object is given to the plugin, which contains various services that you may need. It is a good idea to keep a pointer to this object for further use.

Once instantiated, the plugin will be queried on its UUID and type. It is important that you return a unique UUID for your plugin. This identifier will be stored into the Hopper databases, and the UUID will be the only reference to find the correct CPU support for instance.

File Loader Architecture

Detecting the file type

A *File Loader Plugin* conforms to the `FileLoader` protocol.

When one loads a file into Hopper, each file loader plugin is consulted using the `-[FileLoader detectedTypesForData:]` selector. The plugin's task is to analyse the given `NSData` object, and to return an array of `DetectedFileType` objects. For instance, it's possible for a MachO file loader that parses a Fat MachO file to return as many `DetectedFileType` objects as architectures found in the file.

The `DetectedFileType` object is really simple; you can instantiate an empty one using the `-[HopperServices detectedType]` method of the `HopperServices` object you received during the initialisation phase of the plugin. Your role is to set up as many fields as possible, like the textual representation that will be given to the user (the `fileDescription` property), etc.

Three properties need a special attention: they are used to give an information about how to find the CPU plugin that will be needed to parse the file content. These properties are `cpuFamily`, `cpuSubFamily` and `addressWidth`. The later is used to inform Hopper if the file should be treated as a 32 bits or a 64 bits file (at present, it is not possible to return anything else than 32 or 64). The `cpuFamily` and `cpuSubFamily` are hints used to find the right CPU plugin to use. Hopper is shipped with CPU plugins for the *intel*, *arm* and *aarch64* families.

- For the *intel* family, the supported subfamilies are *x86* and *x86_64*.
- For the *arm* family, the supported subfamilies are *v6*, *v7*, *v7s* and *v7m*.
- For the *aarch64* family, the supported subfamily is *generic*.

The `cpuSubFamily` field is considered optional by Hopper.

If your plugin is chosen to load the file, you'll receive in parameter the object you sent. Thus, you can use the `internalID` field to keep information between these two phases.

If the loader needs an interaction with the user, you can set the `additionalParameters` property to an array of `LoaderOptionComponents` objects. These options are:

- An address field, created with `-[HopperServices addressComponentWithLabel:]`,
- A checkbox button, created with `-[HopperServices checkboxComponentWithLabel:]`,
- Or a CPU selector, created with `-[HopperServices cpuComponentWithLabel:]`

(please, don't create more than one CPU selector by DetectedFileType object).

In each case, the label is just a textual label for the user interface. If the loader is selected, the `LoaderOptionComponents` objects are set up with the value selected by the user. You have access to these values using the properties defined by the `HPLoaderOptionComponents` protocol.

The `lowPriority` property is used, for instance, by the *Raw Binary File Loader* plugin to make sure that it'll appear after any other dedicated plugin. You can use this boolean if you want to appear in the list, but clearly show that you are not dedicated to this file format.

Loading the File

Once the user made its choice, your plugin may be called again to load the file. Its role is then to create the segments and sections, in the same way that an operating system would do. In order to help the plugin, one of the `DetectedFileType` object, created during the detection phase, is given as an argument, where the various additional parameters have been set up.

The central piece of a Hopper document is the `DisassembledFile` object. It's composed of `Segments`, which are the various parts of the executable file that will be mapped into memory. These segments are split into `Sections`.

During the loading phase, you receive an empty `DisassembleFile` object, and you need to create the segments and the sections. The process is not very difficult, and you can create them by using the `addSegmentAt:size:` method of the `HPDisassembledFile` protocol, or the `addSectionAt:size:` method of the `HPSegment` protocol. For each part of the executable file mapped into memory, you should create an `NSData` object, and affect it to the segment, using the `-[HPSegment segMappedData:]` method.

You may also need to set some attributes of the `DisassembledFile` object, like:

- The `cpuFamily`, `cpuSubFamily` and `addressSpaceWidthInBits` properties (*this is required*),
- The entry point, using `-[HPDisassembledFile addEntryPoint:]` method,
- Some hint about the detected procedures (if any), using `-[HPDisassembledFile addPotentialProcedure:]` method,
- Labels, using methods like `-[HPDisassembledFile setName:forVirtualAddress:]`,
- Comments, etc.

i Important note: If you need to read bytes from the file, please avoid using the methods `-[HPDisassembledFile readXXXAtVirtualAddress:]`: indeed, these methods use the CPU plugin attached to the file to handle the endianness of the processor, but no plugin is attached at this stage yet! For instance, the Motorola CPU sample (provided with the SDK), uses the `OSRead{Little/Big}IntXX(...)` system methods to read values directly from the data object given to the plugin.

If the `DetectedFileType` object you returned set the `debugData` boolean, this is the `loadDebugData:forFile:usingCallback:` method that will be called, with an already loaded file.

Please note that you can (and should) use the provided callback block to give feedback to the user on the loading process. You can simply call the block with a textual information (the *desc* parameter), and a float value (between 0.0 and 1.0) to show the loading progress.

CPU Support Architecture

Writing a CPU plugin is a little more complicated; at least, it depends on the level of detail that you want to reach.

A CPU plugin is composed of two classes: the main class must conform to the `CPUDefinition` protocol. This object will provide some information about the CPU, like the register names, and so on...

During the file analysis, Hopper will ask to the plugin to create instances of objects which conforms to the `CPUContext` protocol. The context object will have to disassemble instructions, and provide heuristic information to the procedure analysis.

The crucial object is the `DisasmStruct` structure: when Hopper needs to disassemble an instruction in memory, it asks to the `CPUDefinition` to create a `CPUContext`, then it asks to the `CPUContext` to fill a `DisasmStruct` with the default values (usually, you'll only fill the structure with zeroes), then Hopper set the `virtualAddr` and `bytes` fields of the structure, and calls the `-[CPUContext disassembleSingleInstruction:usingProcessorMode:]` method. The method returns the length, in bytes, of the decoded instruction, or `DISASM_UNKNOWN_OPCODE` if the instruction cannot be decoded. The method should also fill the fields of the `DisasmStruct` with information about the operands, etc.

It is very important that you set, at least, the `DisasmStruct.instruction.branchType` field: this field allows Hopper to follow the code during the procedure analysis. If the instruction is known to change the control flow (*ie*, is the `branchType` field has been set to anything else than `DISASM_BRANCH_NONE`), you'll be able to provide this information when Hopper calls your `-[CPUContext performBranchAnalysis:computingNextAddress:andBranches:forProcedure:basicBlock:ofSegment:callsites]` method. This method should set the `next` argument to the address of the instruction that will be executed after this instruction (it's pre-filled with the address after the current address), and the `branches` array with addresses of the possible destinations of the instruction *inside the procedure*. For the jumps outside of the currently analysed procedure, there is an additional array of addresses: the `callsitesAddresses` argument. For instance, here is how the Intel CPU plugin works:

- If the instruction is something like *jne, jeq,...*, the CPU plugin sets the `next` argument to the address after the instruction, and adds the destination operand's value to the `branches` array,
- If the instruction is *jmp*, Hopper analyses the destination address; if the destination *looks like* a method's prolog, the destination address is added to the `callsitesAddresses` argument, else, it is added to the `branches` argument. The `next` argument is set to `BAD_ADDRESS`.
- If the instruction is *call*, the address is added to the `branches` argument, and the `next` argument is set to the address after the instruction.

Once you have decoded an instruction, you should fill the `operand` array of the `DisasmStruct` structure. For each operand, you should set its type to a combination of `DISASM_OPERAND_XXX` macros. According to the type you have set, you may also fill some other parts of the operand structure:

- if the operand is a memory access, you should set the `type` to `DISASM_OPERAND_MEMORY`, and fill the `memory` structure with information about the access,
- if the operand is an immediate value, you should set the `type` to `DISASM_OPERAND_CONSTANT_TYPE`, and set the `immediateValue` field,
- for registers, the type should be set to the logical *OR* of `DISASM_OPERAND_REGISTER_TYPE`, the register class like `DISASM_OPERAND_GENERAL_REG`, and the register index like `DISASM_REG3` for instance.

If you want to refine the analysis performed by your plugin, you should take a look at the `CPUContext.h` file, where you'll find a description of each method your plugin can implement. Once again, take a look at the minimal Motorola 68000 CPU support provided with the SDK archive for implementation details.

Tools

A **Tool Plugin** is the easiest plugin you can write. It will take the form of a menu entry into Hopper, that the user can trigger.

The only method that you'll need to write is the `-[HopperTool toolMenuDescription]` method. It returns a description of the menus to be inserted into the user interface.

In the sample projects, you'll find the implementation of such a method.