# How to use Software Heritage for archiving and referencing your source code: guidelines and walkthrough

Roberto Di Cosmo

# How to use Software Heritage for archiving and referencing your source code: guidelines and walkthrough

Roberto Di Cosmo
Inria, Software Heritage, University of Paris, France
roberto@dicosmo.org

September 2019

Software source code is *an essential research output*, and there is a growing general awareness of its importance for supporting the research process [4, 14, 11]. Many research communities strongly encourage making the source code of the artefact available by archiving it in publicly-accessible long-term archives. Some have even put in place mechanisms to assess research software, like the *Artefact Evaluation* process introduced in 2011 and now widely adopted by many computer science conferences [5], and the *Artifact Review and Badging* program of the ACM [3].

Software Heritage [9, 1] is a non profit, long term universal archive specifically designed for software source code, and able to store not only a software artifact, but *also its full development history*. It provides the ideal place to *preserve research software artifacts*, and offers powerful mechanisms to *enhance research articles* with precise references to relevant fragments of your source code.

Using Software Heritage for your research software artifacts is straightforward and involves three simple steps, described in the picture below:


In this document we will go through each of these three steps, providing guidelines for making the most out of Software Heritage for your research: Section 1 describes the best practices for preparing your source code for archival; Section 2 shows how to archive your code in Software Heritage; Section 3 shows the rich functionalities you can use for referencing in your article source

code archived in Software Heritage; finally, in the Appendix you will find a formal description of the different kinds of identifiers available for adressing the content archived in Software Heritage.

# 1 Prepare your repository

We assume that your source code is hosted on a repository publicly accessible (Github, Bitbucket, a GitLab instance, an institutional software forge, etc.) using one of the version control systems supported by Software Heritage, currently Subversion, Mercurial and Git [1].

It is highly recommended that you provide, in your source code repository, appropriate information on your research artifact: it will make it more appealing and useful to future users (which might actually be *you* in a few months).

Well established best practice is to include, at the toplevel of your source code tree, three *key files*, README, AUTHORS and LICENSE, with the information described below.

**README** : A description of the software.
This file should contain *at least*

- the name of the software/project
- a brief description of the project.

It is also *highly recommended* to add the following information

- pointers to the project website and documentation,
- pointer to the project development platform,
- license for the project (if not in a separate LICENSE file),
- contact and support information,
- build/installation instructions or a pointer to a file containing them (usually INSTALL)

In could be useful to provide here also some information for the users, like a list of features or informations on how to use the source code

**AUTHORS** : The list of all authors that need to be credited for the current version.
If you want to specify the role of each contributor in this list, we suggest to use the taxonomy of contributors presented in [2], which distinguishes the following roles: *Design, Architecture, Coding, Testing, Debugging, Documentation, Maintenance, Support, Management.*

**LICENSE** : The project license terms.
For Open Source Licenses, it is strongly recommended to use the standard names that can be found on the `https://spdx.org/licenses/` website.

Future users that find your artifact useful might want to give you credit by citing it. To this end, you might want to provide instructions on how you prefer your artifact to be cited. There are many possibilities for doing that, and you might want to also provide structured citation information in specific formats, like CodeMeta (usually in a file named **Codemeta.json2**) or CFF (usually in a file named **CITATION.cff3**).

[1] For up to date information, see `https://archive.softwareheritage.org/browse/origin/save/`

### 1.1 Learning more

The seminal article *Software Release Practice HOWTO* by E. S. Raymond [13] documents best practices and conventions for releasing software that have been well established for decades, and form the basis of most current recommendations. Interesting more recent resources include the REUSE website [10], which provides detailed guidance and tools to verify compliance with the guidelines, as well as [12], which focuses more on research software.

## 2 Save your code

Once your code repository has been properly prepared, you only need to:

- go to `https://archive.softwareheritage.org/browse/origin/save/`,
- pick your version control system in the drop-down list, enter the code repository url [2],
- click on the Submit button (see Figure 1).

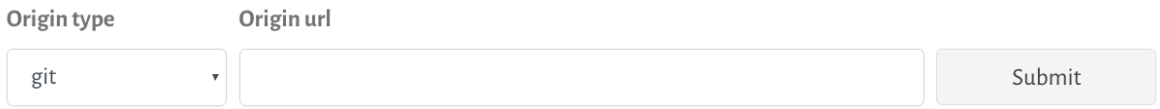



Figure 1: The Save Code Now form

**That's it, it's all done!** No need to create an account or to provide personal information of any kind. If the url you provided is correct, Software Heritage will archive your repository, with its full development history, shortly after. If your repository is hosted on one of the major forges we already know, this process will take just a few hours; if you point to a location we never saw before, it can take longer, as we will need to manually approve it.

**For hackers:** you can also request archival programmatically, using the Software Heritage API [3]; this can be quite handy to integrate, for example, into a Makefile.

## 3 Reference your work

Once your source code has been archived, there are many ways to reference it in your article. We present here three common use cases:

- link to the *full repository* archived in Software Heritage,
- link to a *precise version of the software project*,
- link to a *precise version of a source code file*, down to the level of the line of code.

[2] Make sure to use the clone/checkout url as given by the development platform hosting your code. It can easily be found in the web interface of the development platform.

[3] For details, see `https://archive.softwareheritage.org/api/1/origin/save/`

To make this concrete, in what follows we use as a running example the article *A "minimal disruption" skeleton experiment: seamless map and reduce embedding in OCaml* by Marco Danelutto and Roberto Di Cosmo [6] published in 2012. This article introduced a nifty library for multicore parallel programming that was distributed via the `gitorious.org` collaborative development platform, at `gitorious.org/parmap`. Since Gitorious has been shut down a few years ago, like Google Code and CodePlex, this example is particularly fit to show why pointing to an *archive* that has your code is better than pointing to the collaborative development platform where you developed it.

### 3.1 Full repository

In Software Heritage, we keep track of all the *origins* from which source code has been retrieved, and finding a given `origin` is as easy as adding in front of it the prefix `https://archive.softwareheritage.org/browse/origin`

These origins are the exact *URLs of the version control system* that a developer would use to clone a working repository, and are the same urls that you pass to the *Save Code Now* form described in Section 2.

In our running example, for the Parmap code on *gitorious.org*, this origin is `https://gitorious.org/parmap/parmap.git`, so the URL of the *persistently archived full repository* is the following:

`https://archive.softwareheritage.org/browse/origin/https://gitorious.org/parmap/parmap.git`

Just add this link to your article, and your readers will be able to get hold of the archived copy of your repository even if/when the original development platform goes away (as it has actually happened for `gitorious.org` that has been shut down in 2015).

Your readers can then browse the contents of your repository extensively, delving into its development history, and/or directory structure, down to each single source code file [4].

**N.B.**: if you are unsure about what is the actual origin URL of your repository, you can look it up using the search box that is available at `https://archive.softwareheritage.org/browse/search/`

### 3.2 Specific version

Pointing to the full archived repository is nice, but a version controlled repository usually contains all the history of development of the source code, whiche records different states of the project, usually called *revisions*.

In order to support reproducibility of scientific results, we need to be able to pinpoint precisely the state(s) of the source code used in the article. Software Heritage provides a very easy means of pointing to a precise *revision*, via a standard identifier schema, called SWH-ID, which is fully documented online and is discussed in the article [8].

In our running example, the Parmap article, the exact revision of the source code of the library used therein has the following SWH-ID:

```
swh:1:rev:0064fbd0ad69de205ea6ec6999f3d3895e9442c2;
origin=https://gitorious.org/parmap/parmap.git;
```

[4] For a guided tour see `https://www.softwareheritage.org/2018/09/22/browsing-the-software-heritage-archive-a-guided-tour/`

And you can turn this identifier into a clickable URL by prepending to it the prefix `https://archive.softwareheritage.org/`: you can try it live right now by clicking on this link.

### 3.2.1 Getting your SWH-ID

A very simple way of getting the right SWH-ID is to browse your archived code in Software Heritage, and to navigate to the revision you are interested in. Click then on the *permalinks vertical red tab* that is present on all pages of the archive, and in the tab that opens up you select the *revision* identifier: an example is shown in Figure 2.

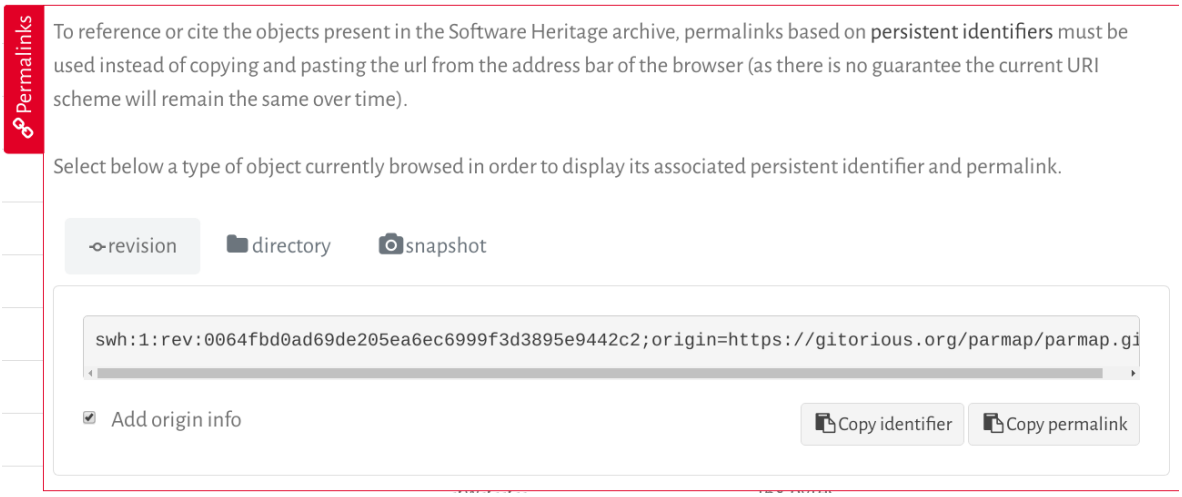



Figure 2: Obtaining a Software Heritage identifier using the permalink box on the archive Web user interface

The two convenient buttons on the botton right allow you to copy the identifiers or the full permalink in the clipboard, to insert in your article as you see fit.

### 3.2.2 Generating and verifying SWH-IDs (for the geeks)

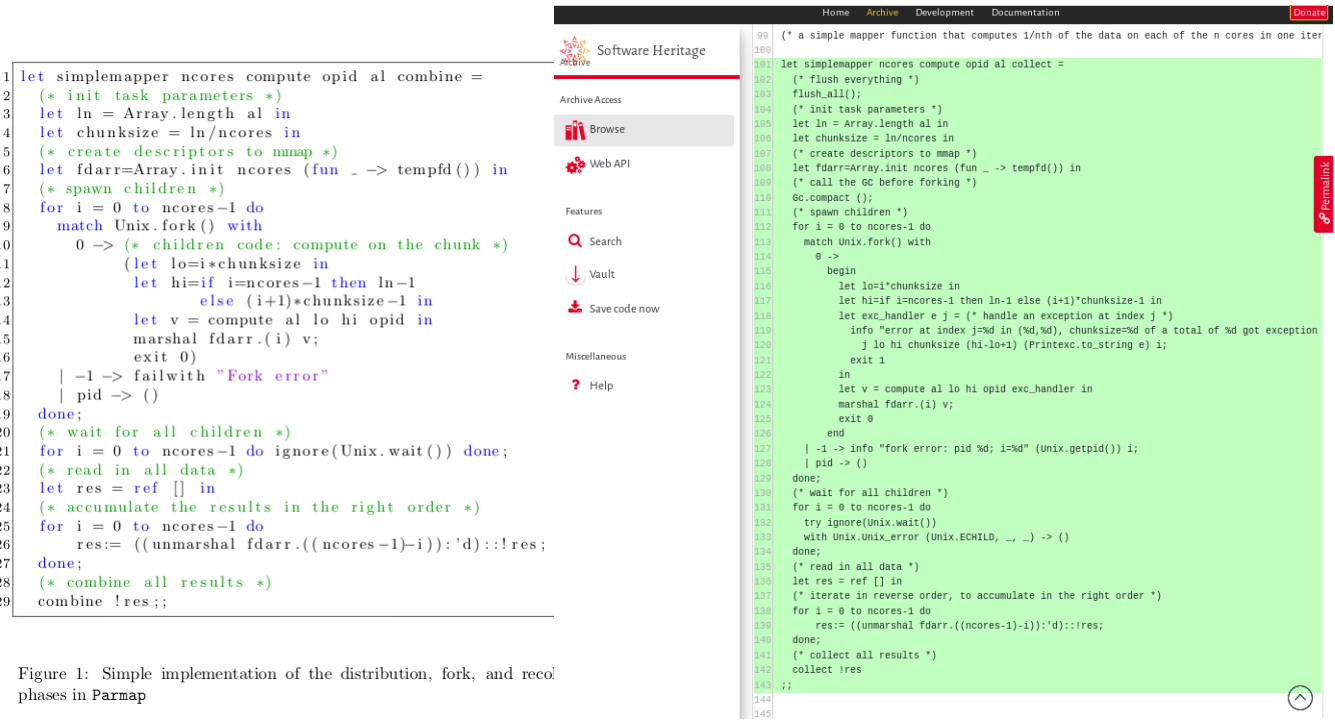
Version 1 of the SWH-IDs uses git-compatible hashes, so if you are using git as a version control system, you can create the right SWH-ID by just prepending `swh:1:rev:` to your commit hash. This might come pretty handy if you plan to automate the generation of the identifiers to be included in your article: you will always have your code and your article in sync!

Software Heritage identifiers can also be generated and verified independently by anyone using `swh-identify`, an open source tool developed by Software Heritage, and distributed via PyPI as `swh.model` (stable version at swh:1:rev:6cab1cc81118877e2105c32b08653509475f3eaa; origin=https://pypi.org/project/swh.model/).

## 3.3 Code fragment

A particularly nifty feature of the SWH-IDs supported by Software Heritage is the ability to pinpoint a fragment of code inside a specific version of a file, by using the `lines=` qualifier available for identifiers that point to files.

Let's see this feature at work in our running example, which shows clearly how an article can be greatly enhanced by providing pointers to code fragments.



```
let simplemapper ncores compute opid al combine =
  (* init task parameters *)
  let ln = Array.length al in
  let chunksize = ln/ncores in
  (* create descriptors to mmap *)
  let fdarr=Array.init ncores (fun _ -> tempfd()) in
  (* spawn children *)
  for i = 0 to ncores-1 do
    match Unix.fork() with
      0 -> (* children code: compute on the chunk *)
           (let lo=i*chunksize in
            let hi=if i=ncores-1 then ln-1
                   else (i+1)*chunksize-1 in
            let v = compute al lo hi opid in
            marshal fdarr.(i) v;
            exit 0)
    | -1 -> failwith "Fork error"
    | pid -> ()
  done;
  (* wait for all children *)
  for i = 0 to ncores-1 do ignore(Unix.wait()) done;
  (* read in all data *)
  let res = ref [] in
  (* accumulate the results in the right order *)
  for i = 0 to ncores-1 do
      res:= ((unmarshal fdarr.((ncores-1)-i)):'d)::!res;
  done;
  (* combine all results *)
  combine !res;;
```

Figure 1: Simple implementation of the distribution, fork, and reco phases in `Parmap`

(a) as presented in the article [6]

(b) as archived in Software Heritage

Figure 3: Code fragment from the published article compared to the content in the Software Heritage archive

In Figure 1 of [6], which is shown here as Figure 3a, the authors want to present the core part of the code implementing the parallel functionality that constitutes the main contribution of their article. The usual approach is to typeset in the article itself *an excerpt of the source code*, and let the reader try to find it by delving into the code repository, which may have evolved in the mean time. Finding the exact matching code can be quite difficult, as the code excerpt is *often edited* a bit with respect to the original, sometimes to drop details that are not relevant for the discussion, and sometimes due to space limitations.

In our case, the article presented 29 lines of code, slightly edited from the 43 actual lines of code in the Parmap library: looking at 3a, one can easily see that some lines have been dropped (102-103, 118-121), one line has been split (117) and several lines simplified (127, 132-133, 137-142).

Using Software Heritage, the authors can do a much better job, because the original code fragment can now be precisely identified by the following Software Heritage identifier:

> swh:1:cnt:d5214ff9562a1fe78db51944506ba48c20de3379;
> origin=https://gitorious.org/parmap/parmap.git;
> lines=101-143

This identifier can also be easily obtained using the permalink box shown in Section 3.2.1 above, and it will **always** point to the code fragment shown in Figure 3b.

The caption of the original article shown in Figure 3a can then be significantly enhanced by incorporating all the clickable links needed to point to the exact source code fragment that has been edited for inclusion in the article, as shown in Figure 4.

> Simple implementation of the distribution, fork, and recollection phases in `Parmap` (slightly simplified from the actual code present in the version of Parmap used for this article)

Figure 4: A caption text with links to code fragment and revision

When clicking on the hyperlinked text in the caption shown above, the reader is brought seamlessly to the Software Heritage archive on a page showing the corresponding source code archived in Software Heritage, with the relevant lines highlighted (see Figure 3b).

For LaTeX users, the caption of 4 can be written using a few convenient auxiliary macros, as shown in Figure 5.

```
\newcommand{\swhurl}[1]{https://archive.softwareheritage.org/#1}
\newcommand{\swhref}[2]{\href{\swhurl{#1}}{#2}}

...

 \caption{Simple implementation of the distribution,
 fork, and recollection phases in \texttt{Parmap}
 (slightly simplified from the
 \swhref{swh:1:cnt:d5214ff9562a1fe78db51944506ba48c20de3379;
         origin=https://gitorious.org/parmap/parmap.git;
         lines=101-143}
        {actual code})
 presented in
 \swhref{swh:1:rev:0064fbd0ad69de205ea6ec6999f3d3895e9442c2;
         origin=https://gitorious.org/parmap/parmap.git}
        {the version of Parmap used in this article}
```

Figure 5: Adding clickable hyperlinks to Software Heritage in LaTeX

## 4 Acknowledgements

These guidelines result from extensive discussions that took place over several years. Special thanks to Alain Girault, Morane Gruenpeter, Julia Lawall, Arnaud Legrand and Nicolas Rougier for their precious feedback on earlier versions of this document.

## References


[1] J.-F. Abramatic, R. Di Cosmo, and S. Zacchiroli. Building the universal archive of source code. *Commun. ACM*, 61(10):29–31, Sept. 2018.

[2] P. Alliez, R. Di Cosmo, B. Guedj, A. Girault, M.-S. Hacid, A. Legrand, and N. P. Rougier. Attributing and Referencing (Research) Software: Best Practices and Outlook from Inria. `https://hal.archives-ouvertes.fr/hal-02135891`, May 2019. submitted.

[3] Association for Computing Machinery. Artifact review and badging. `https://www.acm.org/publications/policies/artifact-review-badging`, Apr 2018. Retrieved April 27th 2019.

[4] C. L. Borgman, J. C. Wallis, and M. S. Mayernik. Who's got the data? interdependencies in science and technology collaborations. *Computer Supported Cooperative Work*, 21(6):485–523, 2012.

[5] B. R. Childers, G. Fursin, S. Krishnamurthi, and A. Zeller. Artifact Evaluation for Publications (Dagstuhl Perspectives Workshop 15452). *Dagstuhl Reports*, 5(11):29–35, 2016.

[6] M. Danelutto and R. Di Cosmo. A "Minimal Disruption" skeleton experiment: Seamless map & reduce embedding in OCaml. *Procedia CS*, 9:1837–1846, 2012.

[7] Q. Dang. Changes in federal information processing standard (fips) 180-4, secure hash standard. *Cryptologia*, 37(1):69–73, 2013.

[8] R. Di Cosmo, M. Gruenpeter, and S. Zacchiroli. Identifiers for digital objects: the case of software source code preservation. In *Proceedings of the 15th International Conference on Digital Preservation, iPRES 2018, Boston, USA*, Sept. 2018. Available from `https://hal.archives-ouvertes.fr/hal-01865790`.

[9] R. Di Cosmo and S. Zacchiroli. Software heritage: Why and how to preserve software source code. In *Proceedings of the 14th International Conference on Digital Preservation, iPRES 2017*, Sept. 2017.

[10] F. S. F. Europe. Reuse software. `https://reuse.software`, Sept. 2019. Accessed on 2019-09-24.

[11] K. Hinsen. Software development for reproducible research. *Computing in Science and Engineering*, 15(4):60–63, 2013.

[12] M. Jackson (ed). Software deposit: What to deposit (version 1.0). `https://softwaresaved.github.io/software-deposit-guidance/WhatToDeposit.html`, Aug 2018. doi:10.5281/zenodo.1327325.

[13] E. S. Raymond. Software release practice howto. `https://www.tldp.org/HOWTO/html_single/Software-Release-Practice-HOWTO/`, Jan 2013. Accessed on 2019-06-05.

[14] V. Stodden, R. J. LeVeque, and I. Mitchell. Reproducible research for scientific computing: Tools and strategies for changing the culture. *Computing in Science and Engineering*, 14(4):13–17, 2012.

Table 1: EBNF grammar of Software Heritage persistent identifiers

```
<identifier> ::= "swh" ":" <scheme_version> ":" <obj_type> ":" <obj_id> ;
<scheme_version> ::= "1" ;
<obj_type> ::=
    "snp"  (* snapshot *)
  | "rel"  (* release *)
  | "rev"  (* revision *)
  | "dir"  (* directory *)
  | "cnt"  (* content *)
  ;
<obj_id> ::= 40 * <hex_digit> ;
           (* intrinsic object id, as hex-encoded SHA1 *)
<hex_digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
              | "a" | "b" | "c" | "d" | "e" | "f" ;
```

# A Appendix: Reference for SWH-ID identifiers

The SWH-ID identifier schema is fully documented online and is discussed in the article [8], but we reproduce here for completeness an excerpt of the documentation.

## A.1 Syntax

Syntactically, persistent identifiers are generated by the `<identifier>` entry point of the EBNF grammar given in Table 1.

## A.2 Semantics

The `swh` prefix makes explicit that these identifiers are related to Software Heritage, and the colon (`:`) is used as separator between the logical parts of identifiers. The scheme version (currently 1) is the current version of this identifier scheme.

A persistent identifier points to a single object, whose type is explicitly captured by `<object_type>`:

**snp** identifiers points to snapshots,

**rel** to releases,

**rev** to revisions,

**dir** to directories,

**cnt** to contents.

The actual object pointed to is identified by the intrinsic identifier `<object_id>`, which is a hex-encoded (using lowercase ASCII characters) SHA1 [7] computed on the content and metadata of the object itself.[5]

[5] See `https://docs.softwareheritage.org/devel/swh-model/persistent-identifiers.html` for more details.

### A.3 Git compatibility

Intrinsic object identifiers for contents, directories, revisions, and releases are, at present, compatible with the Git way of computing identifiers for its objects. A Software Heritage content identifier will be identical to a Git blob identifier of any file with the same content, a Software Heritage revision identifier will be identical to the corresponding Git commit identifier, etc. This is not the case for snapshot identifiers as Git doesn't have a corresponding object type. Git compatibility is incidental and is not guaranteed to be maintained in future versions of this scheme (or Git), but is a convenient feature for developers, for the time being.

### A.4 Examples

The identifiers below are all interesting examples of what the Software Heritage identifiers look like.

They are resolved by the Software Heritage browsing pages available at:
`https://archive.softwareheritage.org/<identifier>`

```
swh:1:cnt:94a9ed024d3859793618152ea559a168bbcbb5e2
```

points to the content of a file containing the full text of the GPL3 license

```
swh:1:dir:d198bc9d7a6bcf6db04f476d29314f157507d505
```

points to a directory containing the source code of the Darktable photography application as it was at some point on 4 May 2017

```
swh:1:rev:309cf2674ee7a0749978cf8265ab91a60aea0f7d
```

points to a commit in the development history of Darktable, dated 16 January 2017, that added undo/redo supports for masks

```
swh:1:rel:22ece559cc7cc2364edc5e5593d63ae8bd229f9f
```

points to Darktable release 2.3.0, dated 24 December 2016

```
swh:1:snp:c7c108084bc0bf3d81436bf980b46e98bd338453
```

points to a snapshot of the entire Darktable Git repository taken on 4 May 2017 from GitHub.

### A.5 Contextual information

It is often useful to complement persistent identifiers with contextual information about the object's setting. Currently it is possible to extend the identifier with the optional elements below using the dedicated syntax presented in Table 2:

- the software origin where an object has been found/observed

Table 2: EBNF grammar of complementary contextual information

```
<identifier_with_context> ::= <identifier> [<lines_ctxt>] [<origin_ctxt>] ;
<lines_ctxt> ::= ";" "lines" "=" <line_number> ["-" <line_number>] ;
<origin_ctxt> ::= ";" "origin" "=" <url> ;
<line_number> ::= <dec_digit> + ;
<url> ::= (* RFC 3986 compliant URLs *)  ;
```

- the line number(s) of interest, usually within a content object

The semi-colon (`;`) is used as a separator between the object identifier and other contextual information. Each piece of contextual information is specified as a key/value pair, using the equal sign (=) as a separator. The extended contextual elements should be added in the following manner:

**software origin** a URL where a given object has been found or observed in the wild and used by Software Heritage to ingest the object into the archive.

**line numbers** a single line number or a line range, two numbers separated with the hyphen (`-`). Note that line numbers are purely indicative and are not meant to be stable, as in some degenerate cases (e.g., text files which mix different types of line terminators) it is impossible to resolve them unambiguously.

For example, the following identifier

swh:1:dir:c6f07c2173a458d098de45d4c459a8f1916d900f; origin=https://github.com/id-Software/Quake-III-Arena

points to the source code root directory of the computer game Quake III Arena[6] with the origin URL where it was found; while

swh:1:cnt:41ddb23118f92d7218099a5e7a990cf58f1d07fa; lines=64-72

points to a comment segment with the warning "NOLI SE TANGERE" in a file in the Apollo-11 source code.

[6] See `https://en.wikipedia.org/wiki/Quake_III_Arena`