Working Paper on the legal implication of certain forms of Software Interactions (a.k.a linking)

Release 1 – July 2010

Overview

The aim of this document is to provide some general guidance to lawyers and developers working with free software to understand the technical and (potentially) legal effects of the interaction or interoperation of two programs together.

More specifically, the purpose of this document is to facilitate understanding of different mechanisms of interaction between programs in order to facilitate decision making as to whether a program may or must be considered a *derivative* work of another (original) work, or possibly a *collective* (composite) work incorporating a previous work, or whether it could be considered independent. Even more specifically, we aim to shed some light on the debate regarding the use of GPLv2'd software components, or creating software for GPLv2 platforms, and the scope of the copyleft provisions of this license.

The purpose of the document is descriptive and exploratory, and it does not aim to establish any legal or normative position or "doctrine" in the matter (we leave this for position papers and individual scholarly works). The actual legal effect of any form of interaction will depend on the circumstances of each case, and here we only provide some preliminary (and simplified) examples.

In addition, the legal interpretation and consequences of any form of interaction (e.g. whether it creates a derivative work or not, under which license a program may be distributed, what are the distribution obligations) will also depend on the specific legal framework of the jurisdiction (state) in which the question arises, whether during the course of developing new software or in copyright infringement proceedings. For example, certain jurisdictions may not grant copyright protection for certain aspects or elements of a work (e.g. in the USA, "*processes, systems and methods of operation*") which may limit the scope of exclusive control of a copyright holder.

Finally, due to the nature of the free software environment, the community view – i.e. the opinion of the members of the community from which the software is taken / in which the software is developed - may be of equal if not more relevance than the strict legal interpretation of a license, for the purpose of assessing risks and benefits when taking a decision about the licensing of inter-related components of software.

From a legal perspective, this document is currently mainly aimed at persons working within the EU, though some reference is made to US laws and jurisprudence. Thus the legal framework underlying our comments here consists mainly of the EC Computer Programs Directive (EUCPD, consolidated in Directive 2009/24/EC), other EU copyright directives (Database, Information Society) and national implementations and decisions within the EU. Some definitions are based on the Berne Convention and WIPO Copyright Treaty.

This document is divided into four sections:

- Section 1: Introduction and discussion paper
- Section 2: Discussion on forms of software interaction
- Section 3: Technical Glossary: a glossary of technical concepts and terms (mainly aimed at lawyers) . This tries to describe the main technical jargon that is used by engineers when describing their programs and how they work.

- Section 4: Legal Glossary: a glossary of legal concepts and terms (mainly aimed at software engineers and project leaders). This tries to describe the main legal jargon that lawyers use when trying to analyse the nature of the work, the scope of a persons rights and obligations.

We focus on GPLv2, but much of the analysis could also be applied to GPLv3 – the section on license wording would be slightly different.

"//" denotes "discussion comment".

Terms in *italics* are defined or explained in the Technical and Legal Glossaries at the back.

Introduction

The main issue addressed by this working paper revolves around the following question: does a specific form of software interaction or interoperation create a work that must be distributed under copyleft provisions of GPLv2 (and when does it not)? This question has arisen for two main reasons: first, there is no clear-cut answer to what is a "derivative work", as defined by copyright law, or "work based on another" (and even if there were, this could vary according to jurisdiction) and second, GPLv2 itself is not clear (or rather, has multiple definitions) regarding what it considers falls within the copyleft obligations of redistribution of the whole under the terms of the GPLv2 (Art. 2b in particular). One of the aims of this document is to provide input, in the form of legal analysis and argument, so that a consensus might be achieved on a project by project basis with regard to the interpretation and application of GPLv2 to specific cases of interaction, and the risks involved. Due to the specificities and variety of technologies and software interactions, we do not believe that a general consensus can be achieved across projects and across technologies, however we do indicate where there seems to be some form of agreement within the sector/industry or wider community. For the purpose of this paper, by way of methodology, we break down this issue into five main questions, the first four looking at copyright law and the fifth looking at additional relevant wording of GPLv2.

1. *What is the original software artefact that is being used in the new work, is it protected by copyright, and to what extent?*

   This question raises several issues. The scope of copyright protection is jurisdiction specific. Generally speaking, under the international treaties, works in the public domain and "ideas and principles" underlying the software are not protected (including, under the EU Computer Programs Directive, those that "*underlie its interfaces*" (per Art. 1)). In the second case (ideas and principles) the scope of these concepts is not clearly defined. US legislation (which excludes any "*idea, procedure, process, system, method of operation, concept, principle, or discovery*") and courts (and authors?) seem to have been more active in determining these boundaries (see references below), and have excluded (a) purely functional elements, (b) ideas (when merged with the expression) (c) "scenes a faire", (d) works in the public domain, and (e) facts, among other limits on copyright protection.

2. *When creating and/or distributing the new work including or interacting with the original software artefact, is any act restricted by copyright being performed in relation to that software artefact (i.e. is there a clear infringement: reproduction, transformation, distribution?), and if so, which? (merely copying, or copying and transforming?)*.

   I.e. does creating and redistributing the (combined/inter-related) work involve the performance of an act restricted by copyright, *stricto sensu (*other than distribution of the software artefact itself), regardless of what the GPL may otherwise add. In particular we ask if a particular form of software interaction, under a pure or "bare" copyright law analysis, creates a derivative work of one or both of the interacting software components. This is because the license at least is clear that it applies to derivative works "under copyright law" (here read: strict interpretation of legislation/case law).

   Regarding this question, again we find a difference between jurisdictions regarding the creation of a derivative work or "transformation". While the US law states that a derivative work is a "*a work based upon one or more pre-existing works*" (giving rise to tests of substantial similarity and inclusion and a certain amount of interesting case law), the EUCPD talks of "*the translation, adaptation, arrangement and any other alteration of a computer program and the reproduction of the results thereof*". Within the EU jurisdictions, it seems there is a distinct lack of case law on derivative works of computer programs.

3. *Still within the borders of copyright law, if there is no clear-cut answer to these questions, at what additional test or criteria might a court look to determine if the new work could be considered to be the result of the performance of an act restricted copyright (reproduction or transformation)?* This will be even more case specific. In English law, for example, this may be seen within the context of "non-verbatim copying" or similar tests. Here, we could mention, for example:

   - dependency/independency criteria (does the new work function without the incorporated/inter-related GPL work? Could you swap the GPL component for another one? If so or if not, to what extent? Is there a non-protected API being used as part of the interaction?),
   - "critical functionalities" (does the GPL component provide critical functionalities for the new work – are these functions more than mere "scenes a faire" or "methods", that might be excluded under applicable law?),
   - "made for" (has the plug in been made for a GPL core/kernel, and if so, which part of the core? Does the design of the artefact for which the plug in has been

made exert such an influence on the design and development of the plug in that the second developer is (ab)using the skill and judgement of the first?), or

- "use or reproduction of a substantial part of the skill, labour and judgment invested in the original work" when developing the new work... or
- other such rationale that (relevant) courts may have used in case law (e.g. copyright is also interested in the manner in which a work is created – which is why there are clean room developments - and not just which artefact the work interacts with or what it does once created, so it could look at the development process).

4. *If you have established that the work in question is protected by copyright, and that the act which you are looking to perform is an act restricted by copyright then, irrespective of any purported grant of licence / permission, does the creation or use of the original work amount to fair use, fair dealing or is any other defense available in the relevant jurisdiction?*

   Again, we meet several challenges as the exemptions from copyright infringement vary from jurisdiction to jurisdiction. In the US one would first look to rely on "fair use" or other explicit exemptions (in the UK "fair dealing" exemptions), while in other EU countries legislation tends to have created a series of specific exemption use-cases, most of which are not relevant for our purposes, but usually include exemptions in favour of interoperability. And there may also be a *de minimis* exception, whereby trivial reproduction will not be covered (in England/Wales, extended by exemption for "insubstantial copying").

5. *Having done the "bare" copyright-based analysis, set out in the preceding questions, we can finally ask what, if anything, does the wording of the GPL add to the "bare" copyright-based analysis (particularly if the answer is in the negative, or at least not clear), and how can that wording be interpreted?*

   We know that with a GPL'd work, there will always be an infringement defense (authorisation) prior to distribution, as the license permits reproducing and transformation... however, an important question is: what are the conditions on exploitation of the third party GPL code (or the plug in for the GPL code)? This is because, for instance, the conditions on copying and distribution are different from those on modifying and distribution. To answer this, we would go back to both the answer to question 2 (which copyright act is being infringed?) and the wording of the GPL, and try to resolve any conflicting language.

   This is a key question because the GPL purports to cover not only "works based on the Program" as interpreted by copyright law, but also works that "in whole or in part contain … the Program or any part thereof", leading us to look into the question of collective/composite works (also possibly considered derivative works- not necessarily the result of an "adaptation/transformation", but because of the "inclusion"). One may also need to look at the concept of "work" as understood by the GPL (which or what "work" is the GPL talking about?), which is relevant for Art. 2 of the license. For example, a relevant "work" may be a compiled binary which incorporates a GPL'd library. But a work may also merely contain a GPL'd library because this library is loaded at run-time. Or it may interact with GPL'd dependencies (libraries, whatever) that may be distributed separately (or downloaded separately) but are still required by the new work in order to function (disregarding operating system components, though even that is a question to be answered clearly).

   An interesting and valuable take on the concept of "work" is contributed by the FSF itself, when commenting on "mere aggregation" in its GPL FAQs (see http://www.gnu.org/licenses/old-licenses/gpl-2.0-faq.html#MereAggregation).

   So, we ask if the wording of the GPLv2, in particular, means that the scope of its copyleft provisions apply to the combined work – whether statically, dynamically linked or otherwise related.

   A particular issue with GPLv2 revolves around whether the courts of any jurisdiction would interpret it as having any effect in relation to works which have a connection with a GPL'd work, but are not derivative works as a matter of law or their use otherwise requires the GPL'd code author's consent under copyright law. The wording of GPLv2 is open to interpretation on this point (GPLv3 addresses this point more directly). This issue is best explained by way of example:

   > *A coder takes a work subject to GPL2 ("X"), and incorporates a very small part of it ("P") into another work, ("D"). We have selected P such that incorporation of P in D does not, as a matter of copyright law, require the licence of the original copyright owner of X (this may be because P does not meet the threshold requirements in a particular jurisdiction to attract copyright protection – perhaps it lacks sufficient originality – or because of fair dealing or similar exemptions). It is uncontroversial, that, as a matter of copyright law, the exploitation of D does not require the consent of the copyright owner of*

X (and follows from our definition of "derivative work"). The question, however, is whether such exploitation of D is a breach of the licence under which X itself is exploited. This issue arises from wording in section 2(b) of GPLv2 which refers to a "work...that...contains...the Program or any part thereof".

In other words, by distributing a non-derivative work D, may the coder still be in breach of GPLv2 as it applies to X? So, on a licence basis, could the coder potentially lose her licence to X, if this licence purports to require obligations which restrict otherwise-unrestricted acts, and she performs such an act. The answer to this may depend on whether the terms and conditions of the GPL are considered to be a licence or a contract – and if this has been validly formed, etc., but this may not be the only issue. If the answer to this question is "yes", then we have to look further than copyright law at the relationship between X and D to determine whether the GPL is breached in respect X, if D is distributed other than under the GPL. If the answer is "no", then we only have to consider whether D is a derivative work or otherwise covered by X's copyright rights (in accordance with our definition). So, the question may not necessarily be only "what is one permitted to do by the license in terms of P or D (in our example)?", but also "how does exploitation of P/D affect the licence for X?". Readers should consider this subtle distinction when considering each form of software interaction. Note that questions 2 and 3 overlap, or at least it is difficult and even artificial to separate the answers, certainly in case-based jurisdictions where court decisions also establish the law (as opposed to interpret it).

At each of these stages, the specific interaction at which we are looking could "fall by the wayside" in copyleft terms, as either copyright protection is not granted, or if it is, there is an exemption, or finally, the license itself provides for exemption from copyleft obligations. (// For the sake of discussion: a wider interpretation of the license may rely on the use of functional or factual elements of expression of the GPL'd code, which may run against copyright principles which do not protect these parts, even if they are re-incorporated into the new work).

In addition, we must add that the GPL, as a copyright license, must be interpreted in each jurisdiction under the applicable laws in force (with the additional cross-border complication of determining which law should apply under conflict of law/private international law rules). The US and EU / EU member state laws differ, particularly in their respective formal definitions of "derivative works", "collective works" and "composite/composed works" (see legal glossary at the end of this document). And another layer of complication is created if the GPL is considered a contractual document, to which varying jurisdiction-specific rules of contractual interpretation (*contra proferentem*, intention of the parties, etc.) may apply.

As we comment in the main articles on the different interactions analysed here, it is argued that there are some combinations of software programs that will generally always produce a derivative work, while other forms may not not. But the dividing line between the two is not clear and in fact will depend on the facts of each case. For example, one of the major arguments in this area has been that dynamic linking – which does not involve a transformation or compilation/linking of the linked code/library at development or build time - does not necessarily create a derivative work of that code/library, and the external library is only reproduced and distributed (Art. 1, GPLv2), rather than transformed and distributed (Art. 2, GPLv2). Although it is then reproduced and linked at run-time (which might create a derivative work), this is only created in the user's computer memory, after redistribution. If this is correct, the "strong copyleft" view, in order to apply conditions to the distribution of code dynamically linking to the library, may then have to rely on two arguments:

- "collectivity" (for lack of a better word): the dynamically linked library or plug-in is distributed along with the application code that uses it, as an integral part of the "combined" program, and the linking program is not an "independent and separate" work in itself. In this case, the GPL would apply to all the program that is distributed, not just the GPL'd library. \\what about the position under GPL 3.0?
- "interdependency": the main program that uses the GPL'd library is designed and written to include and use the functionalities of the external library (at runtime) and thus "depends" on the library to work. In this manner, an interdependent compilation has been created, which is argued to fall under the copyleft rules of the GPL

Neither of these is necessarily a strong or definitive argument, as the new code could be written to a public API and use the GPL'd library as an implementation (among others) of that API. In addition, writing code to use a library (and then executing the library at runtime) could be considered merely "using" the library in the intended manner covered by fair use (in the USA), as well as specifically excluded from the GPLv2 license conditions when it says: "*the act of running the Program is not restricted*" - thus requiring merely a consent to reproduce (but not modify) and redistribute the artefact. This argument has been set out in the main article on dynamic linking.

The "dependency" argument is of interest. It has been argued that if the new program is specifically designed and written to work (only) with certain libraries (or vice versa,

it is designed to be part of an existing third party program, (e.g. like a plug-in), and has little if no other use in any other context), then the program should indeed be considered "based on" (in a contractual meaning, if not a copyright meaning) the third party work. Against this argument, if, in the new work, one could substitute a third party library with another (older, newer, another operating system function, whatever), then it is more likely that the new work would be considered independent of the third party component (and thus either not derivative, or excluded by the "independent" wording of the GPLv2).

So in all events the questions of separation, as regards functionalities, design and architecture, etc., and independence between programs / components both at design and development time are relevant questions and, while only based on hypothetical cases provided by our technical colleagues, we try to look at them in each case.

Subject to the issues above relating to scope, this document only considers **primary infringement**. In other words, potential infringement of copyright by reproduction, transformation (including translation, adaptation and arrangement) and distribution to the public (as set out in Article 4 of EUCPD). In certain jurisdictions, **secondary infringement** of copyright is also unlawful (for example, in the United Kingdom, the Copyright, Designs and Patents Act 1988, section 16(2) provides that copyright is infringed by someone who "without the licence of the copyright owner...authorises another to do...any of the acts restricted by copyright". Other jurisdictions have similar provisions). Thanks, by and large, to litigation from rights owners of music and video content who are seeking to prevent the unauthorised distribution of their material by claiming secondary infringement against entities facilitating the unlawful distribution (but who do not themselves distribute – such as holders of peer-to-peer indices) the scope of secondary infringement at law is constantly changing.

> *It has been argued, for example, that distributing the Linux kernel together with an NDISWrapper amounts to secondary infringement. (An NDISWrapper in this case, is a driver which acts as an interface between the Linux kernel and Microsoft's Windows XP Driver Model interface, and enables hardware for which a Windows XP driver is available, but not a Linux driver, to work with Linux, by using the Windows XP driver instead of the native Windows driver). The code of NDISWrapper is released under GPLv2. The argument runs along the lines that, where the Windows XP driver is not available under the GPL, the mere use of NDISWrapper somehow authorises a breach of GPLv2 as applicable to the kernel, in that it allows the Windows XP Driver to be interfaced (dynamically, as it happens) to kernel code, and that authorisation of that breach is, therefore, a secondary infringement.*

We do not express any opinion as to the validity (or otherwise) of that argument, and reiterate that this document is only concerned with primary infringement. From our example, NDISWrapper itself, in the context of this document, and so far as primary infringement is concerned, can be analysed both from the perspective of a kernel module, and, to the extent that it (potentially) interfaces with non-GPL code, through its interaction with the Windows XP driver dynamic linking. These issues are dealt with in the main section on software interactions. Shims (which are pieces of code that are themselves typically released under the GPL, but act as an interface to non-GPL code, and of which the NDISWrapper is a specialised example) should be considered similarly.

Next steps

As a next stage for this work, it would be interesting to take several specific "real world" cases of interaction to test the hypotheticals postulated here; and expand the analysis with respect to GPLv3.

Further reading (note, many of these articles focus on US law as there has been little serious discussion in the EU on this topic):

- Burk, Dan L.: 'Method and Madness in Copyright Law'. Minnesota Legal Studies Research Paper No. 07-34. Available at SSRN: http://ssrn.com/abstract=999433
- Determann, Lothar (2006): 'Dangerous Liasons--Software Combinations as Derivative Works? Distribution, Installation, and Execution of Linked Programs Under Copyright Law, Commercial Licenses, and the GPL' Berkeley Technology Law Journal, Volume 21, issue 4; online at http://www.btlj.org/data/articles/21_04_03.pdf
- Free Software Foundation: "Frequently Asked Questions about the GPL licenses"; online at http://www.gnu.org/licenses/gpl-faq.html and http://www.gnu.org/licenses/old-licenses/gpl-2.0-faq.html
- Katz, Andrew (2007): "GPL - The Linking Debate", SCL Magazine Vol 18 Issue 3; online at http://www.moorcrofts.com/documents/GPL%20-%20the%20Linking%20Debate.pdf

- Omar Johnny, Marc Miller, Mark Webbink (2010): 'Copyright in Open Source Software – Understanding the Boundaries' IFOSSLR Vol2 Nº1, online at http://www.ifosslr.org/ifosslr/article/view/30
- Ravicher, Dan (2002): 'Software Derivative Work: A Circuit Dependent Determination', Linux.com; online at http://www.linux.com/archive/feature/113252
- Rosen, L (2001): 'The unreasonable fear of infection'. *Linux Journal;* online at *www.rosenlaw.com/html/GPL.PDF*.
- Samuelson, Pamela (2007): 'Why Copyright Law Excludes Systems and Processes from the Scope of its Protection'. Texas Law Review, Vol. 85, No. 1, 2007; UC Berkeley Public Law Research Paper No. 1002666. Available at SSRN: http://ssrn.com/abstract=1002666
- Välimäki, Mikko (2005): 'GNU General Public License and the Distribution of Derivative Works'. The Journal of Information, Law and Technology (JILT) 2005(1), online at http://www2.warwick.ac.uk/fac/soc/law/elj/jilt/2005_1/valimaki/

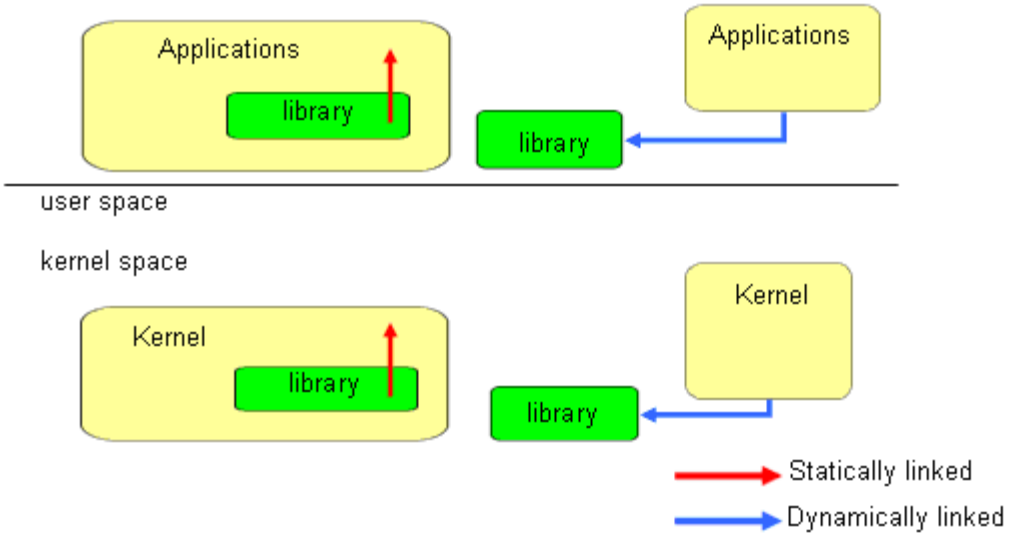
Significant informal debate has been made on this topic, some learned, some not so. e.g.:

- http://kerneltrap.org/node/1735
- http://kerneltrap.org/Linux/NDISwrapper_and_the_GPL

Methods of of computer program interaction or inter-operation

Static Linking	
Description	In static linking, after the source code is compiled into *object files*, a *linker* will combine these object files into one *executable* at build time (build time linking, as opposed to load time linking). Basically, the linker will copy into the executable the required instructions, data and other symbols of the linked file (and any further object files on which this linked file depends). This one executable will contain the machine code of all the components of the programs that were included in the *link* step (this is called "resolving the dependencies", by automatic inclusion from external files or *libraries* in order to satisfy dependencies between the core program and the libraries). Aspects of static linking: • The resulting executable can be distributed by itself and does not need any shared libraries in order to function. • A statically linked executable may in fact have some external dependencies unresolved at build time, e.g. to system functions or libraries, and these will be resolved and loaded at load or runtime. • Statically linked executables are bigger (file size) than a comparable dynamically linked executable because of the inclusion of all machine code that might eventually be executed.
Legal view(s)	
Preliminary analysis	An executable is generally considered (by the legal community interested in FOSS) to be a *derivative work* of the programs and libraries contained in the executable – i.e. those that are statically linked into the executable, mainly because this is done by way of reproduction and transformation of those components. Answering the 5 key questions: 1. The statically linked library is protected by copyright. As a whole, this would include its header information. 2. Copyright in the static library is indeed infringed in the static linking and redistribution of the library through reproduction and arguably modification of the library. The reproduction right is certainly relevant, and arguably the transformation right, it being argued that linking and compiling the library into the executable creates a derivative work of the library (see below). The distribution right is involved, as the library code would be redistributed as part of the executable. 3. Arguably n/a as we fall clearly within copyright under question 1. See discussion below on wider application of copyright law. 4. By incorporating the whole library, no specific exemption may be available (though there have been arguments that even this form of combining is still only "using" the work as contemplated by the author, thus the interaction could be considered fair use (to be read in the light of the GPL, see next questions). Under a free software license, any user has permission to carry out these acts subject to compliance with applicable obligations in the event of distribution. 5. If the library has a copyleft license such as the GPL, the obligations as to redistribution depend on whether one considers that libraries are merely reproduced in the executable or are (also) transformed (see discussion). As we have noted, it is generally thought that the process of static linking transforms the library, and thus creates a derivative work. Even if this is not the case, there are also arguments to say that the resulting executable "contains the library" (a collective work/ compilation?) and thus the executable is

	a “work based on the program” subject to copyleft obligations or Art. 2 GPLv2 on redistribution. In these circumstances, in order to be permitted to redistribute the GPL library, Art 2 GPLv2 requires the whole work (the executable) to be redistributed under the GPL. This is further reinforced by the expressed intent of the GPL “*to control the distribution of **derivative or collective** works based on the Program*”. As an executable with statically linked libraries contains code from those libraries, it is generally thought that this executable should, when distributed, be licensed under the GPL. Legal appreciations vary: Whether the original source code of the linking file (i.e prior to link time) is a derivative work of a statically linked library can be questioned. All it does is reference external required code (symbols, header information of the library), it does not reproduce the linked code in any manner nor does it transform it. However, there is an argument that, as this source file is designed and written to work with the external library code, it is then dependent on or “based on” the external library (i.e it is not independent). This argument is in turn opposed by a counter argument that the linking program depends more on the interface specifications of the library and/or on the functionalities of the linked library which in both case are arguably not protected by copyright laws (maybe a stronger argument in the US than in the EU, excluding “*procedures, processes, systems, methods of operation*”). In this view, creating symbols to refer and thus link to these functionalities is not an act restricted in any way by copyright and/or covered by fair use. Nor would the symbols that are used to create the link be protected, being interoperability information (interface). A further argument holds that statically linking the code of a library to create an executable is the expected and normal “use” (fair use?) of a library and thus creating the executable does not entail transforming the library in any way – merely reproducing elements of the library in the executable. Thus the copyleft obligations of Art. 2 GPLv2 do not arise, despite the wording as to “containing the library” as there is no modification (a prior requisite for Art. 2), and rather the obligations under Art 1. (copying and distributing) will apply. However, despite these arguments, most lawyers seem to hold that the fact that the library is statically linked, (i.e code is added to the executable), results in a whole that is derivative of the library. Finally, we should look at the scope of the concept “Work” in Art. 2: it could cover not just the new code, but the combined work of new code plus original GPL'd software artefact. It is this “whole” that must be distributed under terms compatible with the GPLv2, including all its component parts. FSF analysis GPL FAQs
Risk/Benefit analysis	Failing to observe the third-party licensing requirements on the contained code/files (e.g. copyleft obligation to distribute the whole work under the terms of the license of the contained code) would give rise to copyright infringement, and/or contract violation, if the GPL is considered a contract.
Common practice	Development time: All sub-components to be used should be under inbound licenses that are compatible with the outbound license on the executable (general rule for all software!) Distribution/License selection: the outbound license on the executable must be compatible with the licenses on the sub-components. Comment: Should all licenses on components be compatible among themselves? A: Arguably yes, if one of the components is GPL, because this obliges the distributor to license the program as a whole (including all the components) under the GPL, in order to be compliant, and then everything incorporated in the program has to be compatible with GPL.
Comments	A rather wide-spread misunderstanding exists, according to which a licensor using third party components must, upon distribution of the whole, automatically license its relevant work (the whole) under a license compatible with the license of those components. e.g. a company using a GPL-

	licensed component with its product/work, when redistributing this product/work under e.g. a proprietary license, must automatically retract and apply GPL to its own product/work. This is not necessarily so: what the licensor is doing by using the proprietary license is infringing the copyright of the rightsholders of the GPL components. In the event of any copyright suit, it is unlikely the licensor would be forced to distribute the whole under a compatible license (what if there are several licenses available?), but the licensor must withdraw the infringing product until he/she does actually choose to distribute it under a compatible license. Note: This is probably the same for any breach of the license (forgetting to put the text, provide source, etc.), and not just license selection.
Community norms	
View	The community sees static linking as a clear-cut way of creating a derivative or collective work covered by copyleft obligations, and thus potentially violating a copyleft license (like the GPL) when not distributing the combined work under that license.
Common practice	General practice is to apply the GPL to any executable that statically includes GPL libraries.
Visual representation	

Dynamic linking	
Description	When files are “dynamically *linked*”, the content of the external (shared) sub-programs or *library* files are not copied into the executable (by a compiler/linker) but are instead included as references to those shared libraries. To be able to execute the program, the shared / external libraries must be somewhere identifiable on the computer system so that the references can be resolved and the libraries “called up” and executed at load or run time. This means that neither the *source code* of the linking program nor the compiled and linked *executable* of that program contains the source or machine code (respectively) of the external library required to execute the combined work: at load-time or run-time, the system component responsible for loading executables into memory (the run-time linker) must also load the requested object files from shared libraries and fix any references. • The executable set up for dynamic linking usually contains **function names** and **variable names** from the shared libraries or associated header files. • The reference from the executable to the shared library is done by way of header files. The content of these files varies depending on the programming language. e.g. in C it is a purely functional description (and as such probably not copyrightable), but in C++ there could also be code in these header files. In C, the header file is only for the compiler to ensure that you are invoking the function in the right way (plus documentation for the developer) and these files will not be present in the final distributable binary. For C++, the header files can be used in the C mode or if they include source code, these will be linked in a way similar to static linking meaning that the code will be included in the executable. **Comments**: Certain aspects of dynamic linking: • “Dynamically linked executables” are smaller than the equivalent statically linked executables because the "shared code" is not necessarily distributed together with the executable. • When distributing a dynamically linked executable, one must take care to also distribute the required shared libraries, unless these are expected to be present already on the target system (e.g: the GlibC shared libraries). • Using dynamic linking uses less memory because object files pulled in from common shared libraries are present only once in the system's RAM, regardless of how many executables reference that object file. *//distinguish or comment interpreted language scenarios*
Legal view(s)	
Preliminary analysis	It is a point of controversy whether *dynamic linking* differs significantly from *static linking*, and whether dynamically linking shared libraries creates a derivative work of these libraries or, in GPLv2 terms, a “*work based on*” the library. Answering the 5 key questions: 1. See static linking. The difference here may be that only the header file information is “used” at development time, and this may be excluded from copyright protection (including specifically interface information – see EUCPD). It is unlikely that function names and variables referencing the shared library are covered by copyright, however with more sophisticated header information certain protected code may be incorporated in the linking program. 2. (Strict copyright interpretation): Usually, any significant / substantial protected code of the dynamic library is not copied into the executable. In addition, the library may or may not be redistributed with the application (e.g. system libraries, separately distributed dynamic libraries). In the event of redistribution of the library, the copyright in the dynamic library would be infringed: the reproduction right is relevant during creation,

as the developer will copy the library in the development environment and build the application for testing. The distribution right will also be exercised, if the library is included in the distro (even if not compiled with the main program). However, strictly speaking, there may be no "transformation" of the work (but see 3. next). On a strict interpretation of the concept of "transformation", it seems agreed that transformation or modification will happen at load/run time, which may mean that the distributor does not, and the end user does, create the derivative work (if it would be a derivative and not e.g. a mere copy.)

3. (wider court interpretation) In a manner similar to static linking, the transformation right may be involved, because it may argued that building code to use the library and executing it (in load/run time) could create a derivative work of (US "based on", or EU "adaptation of") the library. See introduction for other arguments on dependency.
4. (avoiding infringement) Use of the library in this manner may be considered normal use intended by the author, without prejudice to him/her. Thus this use may fall under a form of "fair use", depending on jurisdiction. Under a free software license, any user has permission to carry out these acts.
5. If the library has a copyleft license such as the GPL, the obligations as to distribution of the linking program depend on whether one considers that shared libraries are (a) merely reproduced and distributed, (b) transformed in some manner and distributed or (c) arranged into a collective / composed work and distributed.

Thus, if we overcome the "not protected by copyright" hurdle, one of the main questions to consider (on a case by case basis) is whether the dynamically linking program

- Is merely "using" the library in the normal manner (i.e. there is only reproduction and "use" by the end user of the linked library).
- Is a *derivative work* of the library (i.e. by adaptation, transformation or arrangement of the library, or is otherwise a "work based on the program"). This will depend on the concept and scope of "derivative" work in the relevant jurisdiction.
- When distributed together with the linked library (when this is required), is deemed a "collective / composed work" containing the linked library, and the linking program is not an "independent and separate work in itself". I.e, regardless whether the use of the dynamic library creates a *derivative work* in copyright terms, is the new work still covered by Art. 2, GPLv2 because the it falls under the (ambiguous) wording of that clause: a "work containing the [library] or a portion of it" or a "*collective work".*

Arguments also hold that it may be irrelevant whether the library is linked at compile/link time or interpretation/run-time, as the resulting image (in RAM) is a derivative work of the library. In this case, the output is created in the computer memory at runtime and executed from there. But then, in the event of dynamic linking, the RAM image is not distributed, so Art. 2B, GPLv2 may not be applicable on distribution of the linking program. A significant difference is "who creates" the combined program (developer, end user) and "where" (build time in the developer's equipment, load/run time in the user's equipment).

It is also argued that this question could be further divided, depending on the type of shared libraries that are used:
1. Situations where an application was written with no knowledge of a library (which was possibly created later than the application). - See also the section on plug-ins, which essentially are created after the application was created .
2. Situations where the shared libraries typically exist on target systems (such as shared libraries distributed together with operating systems).
3. Situations where the dynamically linking program can use multiple libraries as alternatives, or can run without a given library, or can substitute a given library with another, or can run without a given library with the respective function disabled.
4. Situations where the shared libraries do not exist on target systems and where the program is written to work with a non-system specific shared library that has to be copied into the system so that the program can work. There are several possible cases:
    4.a: The library is distributed together with the linking application (it being argued that the only real difference with static linking is the packaging: 2 files are distributed as opposed to 1).
    4.b: The user downloads the shared library (i.e. they are not distributed together).
    4.c: The library is distributed separately (together but separate).

	There is no clear answer whether an application using a library is derivative work of the library in the above cases. It can be argued that number 1, and probably number 2 are not a derivative work of the library, whereas number 4 may be. Number 3 encompasses many different situations. The difference between 2 (system libraries) and 4 (specific libraries) seems to lie in the presumption that operating system libraries are general purpose libraries that can be used with standard/public interfaces. Also, if an operating system library was written with the purpose of it becoming a part of the operating system, then it was written to work together with many different applications in a standard-like environment. This is supported by the general consensus that it is possible to write computer software applications that are independent (legally speaking) from the operating system – i.e. an application working on an operating system does not create a derivative work of the operating system. Otherwise, many programs using a Windows DLL or shared objects in the Linux kernel would be derivative works of those systems, and there is consensus that this is not the case. However, many dynamic libraries offer public interface information, thus the same argument would apply to case 4. ***Comment****:* It is also probable that the degrees of "separation" and "dependency" are something a court would look at, especially as this is explicitly mentioned in Art. 2, GPLv2: "*If identifiable sections of that work are not derived from the Program, and can be reasonably considered* ***independent and separate works*** *in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works".* Would the two programs (linking and linked files) be considered a "single work"? If dependency between an application and a library is in practice strong (e.g. *function calls to each other and share data structures,* in the words of the FSF FAQ*)*, this would support the "depending program" interpretation, i.e. the application being classified as a derivative work of the library or at least being covered by the GPL copyleft obligations (e.g. as a collective work) (which is what one wants to know in practice). This may also give rise to the argument that only those parts of the work that call upon the shared library are derived, but the rest of the program is independent and thus not covered. Even if they are not "dependent works", would a court consider that substantial reproduction was made (of elements of the shared library) reflecting the skill and judgement of the original author. This argument may be difficult to uphold if the only parts reproduced are interfaces. Strong copyleft arguments may also be denied on the basis that dynamic linking merely "uses" the GPL'd shared library (the GPL saying that the "running of the Program is not restricted"). Finally, when we have to interpret the GPL terms, we come to the oft-repeated debate whether the GPL is a contract, with its terms interpreted *contra proferentem* when there are conflicting provisions; or if it is a mere license under which, for example under continental EU jurisdictions, the courts tend to give stronger protection to the author, in the event of any doubt. Dynamic linking has been touched on by the FSF in its GPL FAQs: • *What legal issues come up if I use GPL-incompatible libraries with GPL software?* At http://www.gnu.org/licenses/old-licenses/gpl-2.0-faq.html#GPLIncompatibleLibs • "*Can I release a non-free program that's designed to load a GPL-covered plug-in?*" At http://www.gnu.org/licenses/old-licenses/gpl-2.0-faq.html#TOCNFUseGPLPlugins • *"I'm writing a Windows application with Microsoft Visual C++ and I will be releasing it under the GPL. Is dynamically linking my program with the Visual C++ run-time library permitted under the GPL?"* at http://www.gnu.org/licenses/old-licenses/gpl-2.0-faq.html#WindowsRuntimeAndGPL • "*If a library is released under the GPL (not the LGPL), does that mean that any program which uses it has to be under the GPL?*" at http://www.gnu.org/licenses/old-licenses/gpl-2.0-faq.html#IfLibraryIsGPL
Practical Risk/Benefit	Potential copyright infringement by dynamically linking GPL'd libraries and not distributing the "whole" under the terms of the GPL; and/or contract violation, if the GPL is considered a contract.

Common practice	Until this point is decided authoritatively, developers/distributors will need to decide on their policies regarding different dynamic linking scenarios. Erring on the side of caution would be to treat dynamic linking as static linking (for purposes of interpreting copyleft licenses).
Community norms	
View	A large proportion of the community sees dynamic linking as being identical to static linking (for purposes of interpreting copyleft licenses). Certainly with respect to loadable LKM that dynamically link to the kernel. Community view that linking to operating system libraries does not render the linking application subject to the terms of the library to which it is being linked. \\Is this copyright law (application interface) or license grant / interpretation? (martin).
Common practice	Treat dynamic linking the same as static linking, however each case should be judged on its merits.
Visual representation	

Example of static versus dynamic linking (on Linux):

```
$ export LD_LIBRARY_PATH=.

$ cat hello.c
hello() { puts("Hello world"); }
$ cat main.c
main() { hello(); return 0; }
$ gcc -c hello.c
$ gcc -c main.c
$ gcc -shared -olibhello.so hello.o
$ gcc -ohello main.o -L. -lhello
$ ldd hello
        libhello.so => ./libhello.so (0xb75e8000)
        libc.so.6 => /lib/tls/libc.so.6 (0xb74a8000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0xb75eb000)

$ ./hello
Hello world

$ gcc -static -ohello-static main.o hello.o
$ ldd hello-static
        not a dynamic executable
$ ./hello-static
Hello world
```

```
$ ls -lh
total 440K
-rwxr-xr-x   1 josv    mtx        4.8K Mar 24 10:03 hello
-rw-r--r--   1 josv    mtx          33 Mar 24 09:56 hello.c
-rw-r--r--   1 josv    mtx         860 Mar 24 10:02 hello.o
-rwxr-xr-x   1 josv    mtx        403K Mar 24 10:04 hello-static
-rwxr-xr-x   1 josv    mtx         4.5K Mar 24 10:03 libhello.so
-rw-r--r--   1 josv    mtx          30 Mar 24 09:57 main.c
-rw-r--r--   1 josv    mtx         780 Mar 24 10:02 main.o

$ rm libhello.so
$ ./hello
./hello: error while loading shared libraries: libhello.so: cannot open shared object file: No such file or directory
```

In this example the source file "hello.c" contains the definition of a function called "hello". The source file "main.c" contains the definition of a function called "main". Both files contain source code in the C programming language. C prescribes that a program starts execution at a function called "main". Both files are compiled using the GNU C compiler (gcc) into object files called "hello.o" and "main.o". The "hello.o" file is then linked into a shared library called "libhello.so" (again using an invocation of "gcc"). "main.o" is then linked with all required object files (including the "libhello.so" shared library) to create the "hello" executable. This executable is dynamically linked; the "ldd" command displays the dependencies on shared libraries. "main.o" is then linked together with "hello.o" and other object files (included from system libraries) to create a statically linked executable called "hello-static". As you can see in the output of "ls" the statically linked executable is considerable bigger (file size). If we remove the shared library "libhello.so", the dynamically linked "hello" executable can no longer be executed.

Remote Procedure Calls	
Description	Remote Procedure Calls (RPCs) are a type of Inter-Process Communication (IPC) and are a technology for client-server computer system architectures. With RPCs, a client (program) can issue requests to a server (program), being a style of programming that hides the inter-machine and inter-process communication (networking) aspects of that call behind an interface that looks like an ordinary (in-process) function or method call. An Interface Definition Language (IDL) file will describe what you can call remotely to the specific server service. RPC technology hides the complexity of the client-server interaction and transparently deals with things like packaging a call's argument list into a network packet, finding live instances of a server, re-throwing server exceptions as client exceptions and converting between datatypes if the client and server run on different architectures. Popular RPC technologies include CORBA, SOAP and SunRPC. Note: RPC is particular kind of Inter-Process Communication (IPC). An IPC calls a function or service in a different process (which may be or not in the same device), so that • RPC will be used to call another device • IPC (generically) will be used within the same device (but not within the same process).
Legal view(s)	
Preliminary analysis	Interaction between a client and a server using RPCs most likely does not have copyright implications in the sense of creating combined or derived works. Answering the 5 key questions (assuming the server to be the pre-existing code - under GPL license): 1. The relevant code that is exploited is the interface information for calling the procedure (IDL file). This may not be covered by copyright. 2. Usually, protected code of the server is not copied into the client. No code, symbols, function names or other interface information from the server is included in the client – the required information regarding what can be called is published in an IDL file. Neither is the server code necessarily redistributed with the client. ◦ Thus the reproduction right is not relevant (except maybe if the server is distributed along with the client - but even then, the nature of the server makes this an independent program from the client). ◦ If the server is not distributed, nor is the distribution right. ◦ The client program is unlikely to be seen as an adaptation (derivative work) or collective work of the server. 3. No permission is required with respect to the server (from a copyright perspective – the RPC will need access privileges). 4. Distribution of the server, if relevant, must be done in accordance with the terms of its license. This will not affect the license on the client (except see below). Using IPC/RPC will, in most cases, simply involve the routine exchange of data between client and server. In copyright terms, there are two separate works and neither is a derivative of the other. So a GPL program and a proprietary program can co-exist and exchange data using these mechanisms without risk of copyleft. In some limited circumstances, we understand that the intimacy, nature of and organisation of the data exchanged could give rise to the two

	programs being viewed as a single work, but this will be the exception rather than the rule, as most IPC/RPC involves well separated and clearly differentiated programs. This same analysis can be applied to simple API based interactions.
Risk/Benefit	RPCs are a useful means of communication between programs that simplifies the licensing obligations between client and server (in the context of non-network oriented licenses like GPL, etc.). Comment: Note that licenses like the Affero GPL (AGPL) use interaction across a network as the defining characteristic for applying its copyleft-style obligations (see section 13 of the AGPL: http://www.gnu.org/licenses/agpl-3.0.txt).
Common practice	No concerns when calling GPL'd server functions Be careful with the copyleft implications of the AGPL in the context of RPCs (though it is thought that plug-in architectures may be used to avoid the copyleft obligations of the AGPL)
Community norms	
View	The community seems not to attach copyright consequences to the use of RPCs.
Common practice	None that we are aware of.
Visual representation	

System calls	
Description	Programs need access to the resources of the computer (such as access to files, memory, CPU, etc.). This access is one of the primary responsibilities of the operating system. Operating systems provide an API that exposes these functionalities to programs. This API is traditionally known as system calls. A system call is the method used by a program to request a service from the operating system. For example, when a program wants to open a file, the runtime libraries of the compiler translate this request into a system call to the operating system. The operating system proceeds to find the file, open it, and return a “handle” to the program. Systems calls tend to be well documented and standardized. For example, POSIX or "Portable Operating System Interface [for Unix]" is the name of a family of related standards specified by the IEEE to define the application programming interface (API) for software compatible with variants of the Unix operating system (although the standard can apply to any operating system) (source: Wikipedia). POSIX documents that the system calls should be supported by a Unix operating system. A program should usually interact with the operating system via its runtime library (e.g. Windows' DLLs). This library interacts with the operating system via system-calls. For example, a C program that wants to open a file uses the runtime library function fopen. The implementation of fopen uses the system call (open).
Legal view(s)	
Preliminary analysis	Answering the 5 key questions (assuming the host operating system to be the pre-existing code - under GPL license): 1. The relevant code that is exploited by the application program is the files / code of the operating system itself and the interface information for making the system call (e.g. POSIX). While the API specification itself may be covered by copyright (as a literary work), using the API is not. The EUCPD specifically excludes interface information / specification from copyright protection as regards the programs using it. 2. Usually, no protected code of the operating system is copied into the application program. Neither is the operating system code necessarily redistributed with the program *(there may be cases where OS libraries are included with an application, to ensure they are on the system so as to be called – e.g. RPMs that call in other packages).* Thus, • the reproduction right is not relevant except maybe if the operating system is distributed along with the application (e.g. in an “appliance”) but even then, the nature of the operating system makes this an independent program from the application. • If the operating system is not distributed, nor is the distribution right. 3. No permission is required with respect to the operating system. 4. Distribution of the operating system, if relevant, must be done in accordance with the terms of its license. This will not affect the license on the client (except see below). The program itself is independent. Linus Torvalds, the original author of the Linux kernel, has stated in the Linux/COPYING file (license applicable to Linux kernel) that programs in “user space” that only use the services of the kernel are not considered derivative works of the Linux kernel. This interpretation (arguably a license term applicable to his code and derivative works thereof) would support any arguments that programs making system calls from “user space” are outside the scope of the Linux kernel GPLv2.

	*'NOTE! This copyright does *not* cover user programs that use kernel services by normal system calls - this is merely considered normal use of the kernel, and does *not* fall under the heading of "derived work"*
Risk/Benefit	
Common practice	The traditional view has been that a program that interacts with the operating system using only system calls is not a derivative work of the operating system, and thus not subject to its licensing terms.
Comments	
Community norms	
View	Other than calls (if any) from plug-ins in the kernel space, higher level programs in "user space" accessing operating system and computer resources are not covered by any copyleft obligations pertaining to the operating system.
Common practice	

Macro and template expansions	
Description	Macros are development methods. Instead of writing an identical passage several times, a macro can be used. Some programming languages provide the ability to expand instructions into more complex ones. For example, a single line of code might be expanded into a large number of lines of source code BEFORE the program is compiled. Two languages that use this method extensively are C and C++. In these languages, the expansion support is provided via two mechanisms: macros and templates. Macros (also called text substitution macros) are simple textual expansions where one or more tokens or symbols in the source code is replaced with others. See example below. In this example, both the macro and the inline function do the same thing. But their mechanisms are different. • The macro is expanded before compilation takes place (a step called preprocessing, done by the preprocessor) • The inline functions are expanded by the compiler during compilation. In both instances, the binary code will include a compiled version of the macro or inlined function. A more complex method for textual expansion is templates. Templates are used to create generic data types and functions. See second example below.
Legal view(s)	
Preliminary analysis	Before or during compilation, macros and template extensions are pulled into the source code, as if they were directly written there. As such code pulled in with macros becomes part of the source code. It should be treated as any code added directly to the source code (a bit like static linking of source code or direct modification, but at a prior stage). Applying the five steps: 1. The relevant code that is exploited is the text/code of the expansion or template. Provided this is original creative work, it will be protected by copyright. 2. The relevant work is specifically copied into the source or pre-processed code of the new work, thus the reproduction right is specifically relevant, and the distribution right will be relevant on distribution of the new work. In addition, it is likely that the new work will be considered a derivative of the expansion, as it consists in an arrangement of (or “based on”) this code. 3. Wider interpretation will only be needed if the expansion code could not be deemed transformed by the original code. Questions of dependency should be fairly clearly answered, as the new code is unlikely to work at all without the expansions. 4. A fair use exception may be put up, saying that macro expansions are specifically written to be reused as such, however this may not be strong as a substantial part of the code of the expansion is reproduced and if specific licensing terms are applied to reproduction and transformation then it it believed these terms will be applied. FOSS licenses allow for this form of exploitation. 5. Macro expansions fall squarely within the concept of being “contained in whole or part” by the new work, thus subjecting the new work to copyleft provisions. Distribution of the new work must be done in accordance with the terms of expansion code license.
Risk/Benefit	There is an additional risk that third-party code with licensing restrictions is inadvertently pulled in. The programmer should be aware of any code included and expanded into the program being developed, and their licensing restrictions. Sometimes these expansions might not be obvious.
Common practice	The code of the macro is reproduced in the source code and redistributed with the application. The license obligations on the macro (and on any third party code pulled in by the macro) would determine the license on the new work.

Community norms	
Risk/Benefit	The perceived risk (in the event of a lack of transparency regarding code pulled in and licensing) might refrain people from using the macro, or the software which embodies the macro.
Common practice	Projects should make the licensing implications of using (expanding) macros and templates clear.

CODE EXAMPLE

For example, this macro finds the maximum of two values:

Definition of the macro:

#define MAX(a,b) (a>b?a:b)

Use of the macro:

printf("The maximum value between 10 and 20 is %d\n", MAX(10,20));

The expanded macro is:

printf("The maximum value between 10 and 20 is %d\n", (10>20?10:20));

A similar expansion mechanism is inline functions in C++. In this case a function is expanded (by the compiler) and inserted in the binary code rather than create a function call to the aforementioned function. For example:

```
inline int MAX(int a, inb b)
{
   return a>b?a:b;
}
```

Template example (slightly modified from Wikipedia):

```
 #include <iostream>

 template <typename T>
 const T& max(const T& x, const T& y)
 {
```

```
  return x > y?x:y;
}

int main()
{
  // This will call max <int> (by argument deduction)
  std::cout << max(3, 7) << std::endl;
  // This will call max<double> (by argument deduction)
  std::cout << max(3.0, 7.0) << std::endl;
  // This type is ambiguous; explicitly instantiate max<double>
  std::cout << max<double>(3, 7.0) << std::endl;
  return 0;
}
```

Plug-ins ***(including LKMs)***	
Description	(See “definitions” section for description). Plug-ins are not really a separate form of software interaction, but a specific type of software artefact that uses software interactions to operate with operating systems or other types of software programs. Below we comment on a couple of methods of interactions commonly used by plug-ins, with an analysis particular to plug-ins. The concept of plug-ins, sometimes called extensions, covers drivers and Linux Kernel Modules (LKMs) and Mozilla Add-ons, just as examples. There is no standard means of interaction between plug-ins and a host system, as there are many forms of plug-ins, however they usually interact with the host by conforming to an interface specification (API) (of the host). Open host APIs provide a standard interface, allowing third parties to create plug-ins, and a stable API allows third-party plug-ins to continue to function as the original host version changes and can extend the life-cycle of (obsolete) applications. Interaction mechanisms: plug-ins may be “invoked” by various methods, including fork and exec (e.g. the fork creating a second process to invoke the execution of the plug-in) and dynamic style linking. • **Machine code plug-ins:** (including LKMs) are loaded using technology akin to "*dynamic linking*", (however rather than the host program “depending on” the plug-in/library, it is usually the other way around: the plug-in depends on the host - see comments 1 and 2 below). • **Script plug-ins**: have to be executed by an interpreter of some programming language, and are intended to be run inside the host system (e.g. Mozilla Firefox add-ons). *[see section on scripts/interpreted languages]* **Comments** **1.** Plug-ins can have bidirectional dependency. While the host calls on the plug-in functionalities (e.g. a device driver), the plug-in itself can also call back into the host environment for system services/functionalities: the host application provides services which the plug-in can use, including a way for plug-ins to register themselves with the host application and a protocol for the exchange of data with plug-ins. Plug-ins depend on the services provided by the host application and do not usually work by themselves – indeed, in most cases, they are written expressly “for” the host (so you have different plug-ins with the same functionalities for Linux, Windows, Solaris, etc.). Conversely, the host application operates independently of the plug-ins, making it possible for end-users to add and update plug-ins dynamically without needing to make changes to the host application. **2.** It is argued that LKMs effectively become part of the running kernel when loaded (leading to the debate on whether LKMs are derivative works of the Linux kernel and thus must be under the GPL, see below). Specifically, LKMs usually run in kernel space and are located in global kernel memory. An LKM cannot resolve its symbols until it is loaded into the kernel, so the LKM remains an ELF object (Executable and Linkable Format). On loading, *symbol* resolution occurs (i.e. calling and loading of external functionalities for this LKM), which can resolve to symbols that are resident in the kernel (compiled into the kernel image) or symbols that are transient (exported from other modules) (see definition of “symbol”).
Legal view(s)	

Preliminary analysis	There is considerable debate about the legal effect of creating and executing a plug-in for a host system, partly due to the different forms of communication and the nature of the plug-in (linked library, “exec”ed executable). Answering the 5 key questions: 1. The relevant code we are looking at is the host: this in itself is protected by copyright. However, it is argued that the plug-in only exploits either (a) interface information or (b) specific functions of the host, which may be under a different (e.g. permissive) license from the host, or else in the public domain. In addition, the symbols, function names and other interface information from the host that may be included in the plug-in may not be excluded from copyright protection. 2. *Strict legislative interpretation*: Usually, protected code of the host is not “copied into” the plug-in. Neither is the host necessarily redistributed with the plug-in, which is usually distributed separately. Thus the reproduction and distribution rights are not relevant except maybe if the host is distributed along with the plug-in (but even then, the nature of the host makes this an independent program from the plug-in). 3. *Broader court interpretation*: Similar to dynamic linking, a court may accept arguments that building plug-in code to extend the host and executing it (in load/run time) creates a derivative work of the host (“based on” or “adaptation of” - see arguments below), thus the transformation right may be involved. 4. *Infringement defense*: [are the host symbol references/interface information “*de minimis*”? - could fair use apply?]. Under a free software license, any user has permission to carry out these acts. 5. *GPL reading*: If the host has a copyleft license such as the GPLv2, the obligations as to distribution of the plug-in depend on whether one considers that host is (a) transformed in some manner by the addition of the plug-in or (b) arranged into a collective work and distributed. However, in neither case is the host usually distributed with the plug-in. Arguments • On the one hand, by interacting with the host system through an (open) API, there are arguments (in line with the “*no protection for procedures, methods of operation, etc*.” argument) that the reach of the copyright on the host goes no further than the API. No reproduction, no transformation. • Certainly with respect to plug-ins that are executed using fork/exec, there seems to be a general consensus that the plug-in is independent of the host and thus can be licensed under any license (see GPL FAQ too). • On the other hand, the dependent interrelation between plug-in and host (mentioned above), both in terms of how the plug-in is developed and how it interacts with the host system, is the basis for arguing that, on loading, the host and plug-in form a single program (a derivative of the host) and the license on the host determines the licensing of the plug-in. [e.g.: the FSF argues that “*Using shared memory to communicate with complex data structures is pretty much equivalent to dynamic linking*” - which, *if they make function calls to each other and share data structures, forms a single program* (see GPL FAQ).]. The trouble with this argument may be that the “single program” is made at run time in the user's hardware, not at any time before. • This dependency argument can be based on several factors (some of which are mentioned in our introductory note), including whether the plug-in was “written for” the host, if they are distributed together or separately (though this should not affect the nature of the plug-in as a derivative work, but the rights involved in distributing the work), the specificity of the interactions with the host (is the code for the MacOS plug-in the same as for the Linux or Windows plug-in?), where the module runs (address space). • “Compiling” the plug-in with the host and distributing the two together is more akin to static linking. Generally speaking, other than LKMs, plug-ins seem to be seen as more “loosely linked” than other external routines such as libraries, which may be due to writing them to standard / public interface. However, if plug-ins are “written” to work with a specific system or parts of that system (e.g. in Linux, operating in what seems to be called “kernel space”), the analysis must be the same as that for static and dynamic libraries. If a specific plug-in is linked dynamically to a host, the reasoning under “Dynamic Linking” should apply. And just like applications that use libraries

with public interfaces, one could argue that a plug-in is written to a standard/public interface of the host.

In practice, often the creator of the host system has made some clarification with respect to licensing issues involved (e.g. Classpath exception). Typically, the host copyright owner indicates that plug-ins are or are not considered a derivative work of the host system or uses a weak copyleft license to allow alternative licensing. For example,

- a plug-in for Eclipse is not considered a derivative work of Eclipse as long as it only uses the plug-in API without any modifications (and thus can be licensed under any terms - see Eclipse FAQ).
- The GIMP follows a different approach, its core libraries (libgimp, and libgimpmath) are available under the LGPLv2 and thus a plug-in can be distributed under any license as long as it honours this license.

However, these statements reflect the views of their authors, rather than necessarily stating or influencing the position at law, although some jurisdictions may consider the intentions of the parties when interpreting a relationship.

See also GPL FAQ: http://www.gnu.org/licenses/old-licenses/gpl-2.0-faq.html#GPLAndPlugins

If a program released under the GPL uses plug-ins, what are the requirements for the licenses of a plug-in?

It depends on how the program invokes its plug-ins. If the program uses fork and exec to invoke plug-ins, then the plug-ins are separate programs, so the license for the main program makes no requirements for them.

If the program dynamically links plug-ins, and they make function calls to each other and share data structures, we believe they form a single program, which must be treated as an extension of both the main program and the plug-ins. This means the plug-ins must be released under the GPL or a GPL-compatible free software license, and that the terms of the GPL must be followed when those plug-ins are distributed.

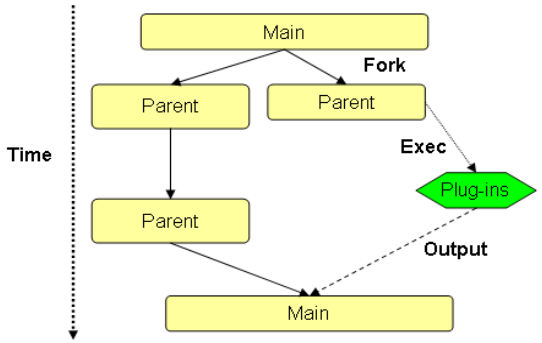
If the program dynamically links plug-ins, but the communication between them is limited to invoking the 'main' function of the plug-in with some options and waiting for it to return, that is a borderline case.

LKM loading

An LKM is argued to be a derivative work of the kernel because it is specifically designed to be part of a whole with the existing Linux kernel. It has no use in any other context, and when deployed, is tightly woven into the rest of Linux. The fact that you typically use Linux header files to compile it is an argument that it is an extension of Linux and subject to the license of those files: a kernel module loaded at run time is essentially the same program you would statically bind into the base kernel if you chose to go that route. If writing a module for the Linux source tree is creating a derivative work, then so must be writing an LKM. LKMs often have to be updated to conform to updates the base kernel. Again, we are not looking at a strict "legislative" application of copyright law, but a wider interpretation (this time made by the Linux Community). To support this, Kernel developers have taken to tagging some symbols that one uses in interfacing an LKM to the base kernel as "GPL-only", thus providing further indication of their intent, should a court wish to read into the license the intent of the licensors. Furthermore, in order for the Linux module loader to let your LKM use these services, the plug-in licensor must include code in the LKM that supposedly certifies that the module is licensed under the GPL. The value of these indications would only be relevant if the court arrives at the stage of needing to interpret the terms of the applicable license.

The argument against this view says that an LKM is something that interacts with Linux (through a defined interface), not something that is part of Linux. The GPL is clear that it applies to works that "are based on" or "contain" part of the original program, and in most cases this is not the

	case. It likens the LKM to a user space program, communicating with the kernel via system calls, or an FTP client program (which would not be a derivative work of any FTP server program). Loading an LKM might be more like plugging an attachment into your vacuum cleaner than like inserting pages into your book. And we know that the blueprints for a vacuum cleaner attachment are not a derivative work of the blueprints for a vacuum cleaner. Note that there are different forms of LKMs, communicating with the kernel core via different methods – some using APIs, other using more direct communications. This may affect the legal analysis, regarding in particular (1) the manner in which the LKM is created – e.g. it may be generically created for POSIX systems; and (2) the elements of the GNU/Linux operating system that are being "used" (linked), which may or may not be protected by copyright.
Risk/Benefit	When compared to dynamic linking it is easier to argue that plug-ins are not a derivative work of the host environment (but an aggregate work).
Common practice	For plug-in writers: If there is only one implementation of the plug-in architecture (one host), contact the host author and ask for a clarification of intent. For host system writers: clarify host licensing intent with respect to the writing and loading of plug-ins.
Community norms	
View	The community seems to see plug-ins as not being derived works of the host (i.e. as being an allowed use of a program not subject to license restrictions) except with regard to plug-ins for the Linux Kernel that run in "kernel space". In a Linux environment, this is supported by the additional "COPYING" statement that goes with the core Linux code.... so long as the plug-in is in user space: plug-ins that are in *user space* and use *standard system call interfaces* get "exempted" by the COPYING exception. Plug-ins that get loaded into kernel space are considered part of the kernel and thus should be licensed under a GPL compatible license. Writers of host-systems usually want plug-ins to be written for their systems so they tend to make it easier to do so [but they may also want plug-ins to be licensed in a certain way].
Common practice	It is not unreasonable to treat this as a low-risk phenomenon. When writing a module and wanting to avoid the risk of application of GPL copyleft obligations, keep them in user space.

Visual representation	

Example with machine code plug-ins:

```
$ export LD_LIBRARY_PATH=.

$ cat gruezi.c
hello() { puts("Gruezi Welt!"); }
$ gcc -ohello2 main.c hello.c gruezi.c
/tmp/ccuafM2s.o(.text+0x0): In function `hello':
: multiple definition of `hello'
/tmp/ccEggN54.o(.text+0x0): first defined here
collect2: ld returned 1 exit status

$ gcc -shared -olibgruezi.so gruezi.c
$ ls -lh libgruezi.so
-rwxr-xr-x    1 josv     mtx          4.5K Mar 24 11:29 libgruezi.so

$ cat main-plugin.c
#include <dlfcn.h>

int main(int argc, char *argv[]) {
  void *handle = dlopen(argv[1], RTLD_LAZY);
  void (*function)() = dlsym(handle, "hello");

  (*function)();
  return 0;
}
```

```
$ gcc -ohello-plugin main-plugin.c  -ldl
$ ldd ./hello-plugin
        libdl.so.2 => /lib/libdl.so.2 (0xb75de000)
        libc.so.6 => /lib/tls/libc.so.6 (0xb74a7000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0xb75eb000)

$ ./hello-plugin libhello.so
Hello world
$ ./hello-plugin libgruezi.so
Gruezi Welt!
```

This example builds on the dynamic linking example (with the "hello" function and shared library "libhello.so"). We first build a shared library called "libgruezi.so" that contains an alternative implementation of the "hello" function. We cannot combine both "hello.o" and "gruezi.o" in one executable because then the function "hello" would be defined multiple times. We then build a new program "hello-plugin" which includes a defintion of "main" (in "main-plugin.c") that uses GlibC functions "dlopen" and "dlsym" to load a plug-in (actually, a shared library), find the "hello" function in the plug-in and then execute it. We can then execute "hello-plugin" with the name of a shared library as its argument and as the example shows we get a different implementation of the "hello" function depending on which plug-in we use.

Interpreted Languages	
Description	Like plug-ins, “interpreted languages” are not a specific form of interaction, however when using these languages (scripts, etc.), interactions arise that may have legal consequences in terms of licensing. Therefore we want to comment briefly on these forms of interaction and explore these consequences. Interpreted languages are computer programming languages where the source file(s) that make up a work are executed by special software (the interpreter, which is typically part of another program) with or without an explicit compile step to translate the source file(s) into machine or byte code. A program written in an interpreted language is usually called a "script". If it is compiled, it will usually be done in real time on the user's computer. Any script must be loaded, interpreted and executed by the interpreter. When it is executed, the script will start making calls to standard libraries (those provided by the scripting language) and other scripts. For example, a Perl script that access a MySQL database is loaded by Perl, and in the process Perl loads the “DBI” and “DBD::Mysql” Perl modules (both Perl scripting libraries), and the MySQL connect libraries (binary libraries, provided by MySQL, that are loaded dynamically or statically). Even a script of few lines might result in a very complex running system. Popular interpreted languages are Perl, PHP, Python and Ruby. Most Unix command line processors (shells) support their own interpreted language which they can execute (shell scripts). Scripts must be distributed in source code form. In few cases, the copyright owner is interested in maintaining the original source code as a trade secret and resorts to obfuscation. The distributed scripts are hence obfuscated versions of the originals. Also, the source code might be subjected to optimization, in order to make the size small and thus suitable for download and processing on-line. Optimization may result in similar results as obfuscation, but the objective is different (trying to achieve small size rather than confidentiality).
Legal view(s)	
Preliminary analysis	Several points need to be considered: 1. interrelationship with the interpreter 2. use of external libraries 3. obfuscation 1. The common view seems to be that programs written in an interpreted programming language are not derived works of the interpreter. An interpreter and a script written in a language are generally separate (copyright) works: neither one is a copy or transformation of the other, nor a collective work including it. Like with a compiler, the interpreter is a separate tool. This means that the script does not need to be released under GPL simply because the interpreter is GPL (this analysis might be different if the script takes in parts of the interpreter, which is not usually the case). When distributing scripts you might also need to distribute the interpreter and thus need to abide by its license in relation to distribution of the interpreter.

	2. To the extent these programs use external libraries (scripts or not) they may well be seen as a derivative work of/work based on or containing those shared libraries that are called up by the interpreter. These shared libraries are usually linked dynamically (see *dynamic linking*). So one of the major concerns with the licensing of scripts is the requirements imposed by other modules required during the execution of the script. However, the combination occurs at load/run time, so it is performed by the end-user and normally not by the distributor and therefore the derivative work, if any, would be created by the end-user For example, in the case above, the mysql connect library is available under the GPLv2 + FOSS exception, and as a consequence the DBD::Mysql module is also licensed in the same way. Any script that uses it might be constrained by this. Comment: In the case of Perl, Larry Wall –its original author- clarifies that "my interpretation of the GNU General Public License is that no Perl script falls under the terms of the GPL unless you explicitly put said script under the terms of the GPL yourself" (see Perl License). 3. Obfuscated code is likely to be considered a copy or derivative work (translation) of the original script. The FSF has commented in various places on interpreted languages in respect of the GPL: • *"If a programming language interpreter is released under the GPL, does that mean programs written to be interpreted by it must be under GPL-compatible licenses?"* At http://www.gnu.org/licenses/old-licenses/gpl-2.0-faq.html#IfInterpreterIsGPL • "*If a programming language interpreter has a license that is incompatible with the GPL, can I run GPL-covered programs on it*?" at http://www.gnu.org/licenses/old-licenses/gpl-2.0-faq.html#InterpreterIncompat •
Risk/Benefit	Generally speaking, see dynamic linking One potential problem is that the writer of a script might not be fully aware of any other modules tha are required to execute the given script (e.g. in the example above, the author only needs to indicate "**require DBI"** (hence might not know that the MySQL connect libraries or the DBD::Mysql modules are required and loaded at execution time). Comment: For derivative works of scripts (originally licensed under a copyleft license) it is probably not acceptable to make the modified scripts available in so-called obfuscated form, or at least your should supply the non-obfuscated source code as "corresponding source code".
Common practice	Be aware that license terms of imported libraries or functions might apply to the scripts that use them.
Community norms	
Risk/Benefit	See dynamic linking and also macro expansions regarding pulled in functions.
Common practice	See dynamic linking

Example interpreted languages:

```
$ cat pi.pl
$V = 10_000_000;

for ($i = 0; $i < $V; $i++) {
  $x = 1 - rand() * 2 ;
  $y = 1 - rand() * 2 ;
  $n++ if sqrt($x * $x + $y * $y) <= 1;
}

print 4 * $n / $V, "\n";
$ perl pi.pl
3.1423732
```

In this example a source file named "pi.pl" contains a script written in the Perl programming language (the script estimates pi using the Monte Carlo method). The "perl" interpreter program is needed to execute the script. In Unix, script files commonly start with a line like "#!/usr/bin/perl" to indicate which interpreter should be used to interpret the script contained in this file.

[*// Develop more complex example*]

Annex 1 – ade's linker examples

STATIC LINKING when building an executable:

Object file A contains a function bar() at address 100 which needs to call up “foo” during its processing:

100: machine code for bar
108: call external function foo()
120: end of bar

Object file B contains a function foo() at address 100 as well (this is where the compiler locates foo; good thing the linker can move things around).

100: machine code for foo
104: end of foo

The linker will combine the two, moving the function foo() from address 100 to some other address so it does not conflict (e.g. to address 150) and then resolve references, leading to

Program AB:

100: machine code for bar
108: call function at address 150
120: end of bar
150: machine code for foo
154: end of foo

Typically a program uses common functions that many other programs use as well: e.g. common graphics routines, or network communication. These common functions are stored as object files, too, but because they are common and (often) shared, we call those object files "libraries" instead. A library works just like other object files (foo, above):

Library C: (a function called snprintf)

100: machine code for snprintf
300: end of snprintf

Now, the linker can do two different things when a program needs the function snprintf() -- it can treat the library C as any other object file and write the contents of C into the resulting program (in the same manner as bar () called for foo(), above). Supposing that function foo() calls snprintf():

Program AB+C:

Parts of the machine code from A
100: machine code for bar
108: call function at address 150

120: end of bar

Parts of the machine code from B
150: machine code for foo
152: call function at address 200
154: end of foo

Parts of the machine code from library C
200: machine code for snprintf
400: end of sprintf

The program on disk now contains a good deal of code from C. At runtime we read the program from disk into memory and that is it.

RUNTIME LINKING, LOADING OR DYNAMIC LINKING (when executing code)

Another approach is to keep the external references as is, but record where they are supposed to come from:

Program AB-C:
Remember at runtime to look at library C!

Parts of the machine code from A
100: machine code for bar
108: call function at address 150
120: end of bar

Parts of the machine code from B
150: machine code for foo
152: call external function snprintf
154: end of foo

the difference here is the line “remember to look up library C” rather than incorporating the code of C.

This yields a smaller program on disk, but at runtime we have to do another linking step: now we read the program from disk into memory, then read the library C from disk into memory, and resolve any other remaining external references. This time around, on disk the program does not contain anything from C, but when executed in memory, it does.

Another example:

```
// This is a function defined within the program itself.
 int usage()
 {
 }

 // This includes an interface definition; one of the parts of the interface
 // is the external function printf(), which lives in a library somewhere.
 #include <stdio.h>

 // Again, a function defined in the program itself.
 int main()
 {
        usage(); // This calls the usage() function defined above.
                // The entire call can be resolved at compile time or
                // as part of the static linking phase, because the
                // function is part of the same object.
        printf(); // This call can be resolved completely by
                // statically linking the library containing printf()
                // into the program, or dynamically at runtime by
                // loading that library into memory and doing the
                // resolution then.
 }
```

Glossary

Technical Definitions	
Byte code (a.k.a. intermediate code)	Byte code is machine code for an idealized (non-existing) processor. Byte code cannot be executed directly by any machine's instruction units and needs to be interpreted with the help of an interpreter or compiled to machine code at runtime (just-in-time compilation). **Comments**: 1. Byte code is protected by copyright in the same manner as machine code. 2. Because byte code instructions are typically very simple (on par with actual machine code instructions) the byte code interpreter can be relatively simple and straightforward. The advantage of using byte code is that porting efforts are restricted to porting the byte code interpreter, after which all programs (even when initially compiled on another platform) run. See also: Virtual Machine
Compiler	A compiler is a computer program (or a set of computer programs) that translate one or more *source files* into *machine or byte code* for a specific (real or virtual) machine. The result of a compile operation is usually one or more *object files*. See also: Object file
Daemon	A daemon is a computer program that runs in the background, rather than under the direct control of a user; they are usually initiated as background processes.
Exec	"Exec" (with a series of specified letters) is a collection of functions that causes a program or process to be executed, with the currently running process to be completely replaced by the program specified in the "exec" command. The exec command can specify environments where the process is to be executed and other variables. Using fork-exec, a process can create a child (fork) and then substitute this child with another process (exec). The original process can either continue processing or wait for the child substitute to return the result of this process. In this manner, two programs can interoperate with each other (also using pipes), the one "calling" the other and sharing information. However, in this case there is no specific combination of code, rather an execution of another process as part of the overall function of the program. **Source**: opengroup.org, wikipedia, GNU bash manual, Cambridge Univ http://www-h.eng.cam.ac.uk/help/tpl/unix/fork.html
Executable	An executable is a file with machine code that can be loaded into memory by the operating system and then executed by the computer/machine/device. The usual extension in Microsoft Windows is ".exe". **Comment**: Executables generally come in two flavours: • *Statically linked executables* are completely self-contained and contain (nearly) all the machine code required to execute the executable's

	function. All dependencies have been resolved by the *linker* at link time and the object files (*static libraries*) declaring the dependencies were included in the executable. • *Dynamically linked executables* contain references to functions in external *shared libraries*. At run time the executable (with help from the operating system) loads required but missing functions from these shared libraries. **See also**: Library, Linker
Fork	Generally, a fork in a multithreading environment means that a thread of execution is duplicated, creating a child thread from the parent thread. The fork operation creates a separate address space for the child. Both the parent and child processes possess the same code segments, but execute independently of each other. When a fork() system call is issued, a copy of all the pages corresponding to the parent process is created, loaded into a separate memory location by the operating system for the child process. **Source**: wikipedia
Header file – Include file	A header or include file often (but not always) defines an interface, commonly containing declarations of variables, subroutines, classes and other identifiers that are used by the instructions of the code in an associated file. Developers who wish to declare standardized identifiers in more than one source file can place such identifiers in a single header file, which other code can then include whenever the header contents are required. The same declarations/identifiers may be found at the front of any file, not necessarily in an external header file. An include file is a source file that is included in another source file by using a special “include” instruction from a compiler or an interpreter. For libraries (delivered as object code) the accompanying header file defines the API that the library exposes to other programs / objects. Without the header file, there would be no useful way to gain access to the library's functionality. Ref: Wikipedia and ade. **Relevance**: System libraries are typically accompanied by header/include files that describe the interface. For standard system libraries, the names of the include files are also standardized. **Comments**: 1. Some header files (C?), can actually include operative computer instructions (code), rather than just a set of declarations. 2. Purely factual data (variable declarations, identifiers) in a header file would generally not have the benefit of copyright protection. (This gives rise to the debate whether including the header file information by dynamic linking causes the linking program to be a derivative of the underlying library – see main articles on linking). 3. Include files typically contain definitions or pieces of code that are needed in multiple source files. Through the process of "file inclusion" these definitions and code "live" in only one location which eases source code maintenance and updates. 4. Include files are often used in C-language programs to define interfaces and macros for use elsewhere. See also: Preprocessor, header file

	// Add reference to "macro expansion" (?) A.k.a ".h file".
Include file	~~An include file is a source file that is included verbatim in another source file by using a special "include" instruction from a compiler or an interpreter.~~ **Comments**: ~~1. Include files typically contain definitions or pieces of code that are needed in multiple source files. Through the process of "file inclusion" these definitions and code "live" in only one location which eases source code maintenance and updates.~~ ~~2. Include files are often used in C-language programs to define interfaces and macros for use elsewhere.~~ ~~See also: Preprocessor, header file~~ ~~*// Add reference to "macro expansion" (?)*~~
Interface (Application Programming Interface, as opposed to "user interface"):	The term "API" or programming interface is used in two manners, being two sides of the same concept. • On the one hand, an API is an abstraction, being a *set of conventions or definitions* to which a developer should develop his/her program so as to access the services (functionalities) of another program (including a library) or whole computer system. The API is thus a conceptual interface to that other program (library, etc.). It is the convention or definition to which input from another program should adhere to so that a program (library, etc.) receiving this input may provide its services. In this manner, the program (library, etc.) enables other developers to create new applications that are interoperable with or can "use" the underlying program, library or system. • On the other hand, the term is used to describe the actual code or part of a program that receives/handles the input from the other programs (as used in the expression "the program exposes its functionalities / accesses the functionalities of X / through its API.") **Comments**: 1. The recitals of the EUCPD, define "interface" as those "*Parts of a program which provide for interconnection and interaction between elements of software and hardware."* This can be read as being either the specification of the code interface (the conventions, conceived as parts of the program) or the actual implementation of those conventions. 2. Another role of an API is to provide an interface between programs with different languages or conventions. Thus its task is to translate parameter lists from one format to another and the interpretation of arguments in one or both directions. In this meaning, the API can be considered a computer program that actually does something (converts or provides facilities for mapping formats, requests from one language to another), rather than just establish a set of definitions/conventions. 3. APIs as a set of conventions would have copyright protection as a literary work. APIs as code would have copyright protection as source code (which is also considered a literary work). However, writing a program "to an API" may not necessarily create a reproduction or transformation of the API (or the code that implements it)... but there is debate whether it would create a "work based on" the API or the program/library that implements the API.

Interpreter	An interpreter is a program that executes a *program* contained in one or more *source files* without the necessity of an explicit *compile* step to translate the source file(s) into *machine code* or *byte code*. E.g. *scripts* are interpreted at run time. **Comment**: Some interpreters perform a partial compilation of the program into an internal representation that facilitates easy execution of the program (e.g.: Google's V8, the Python interpreter, Perl6).
IPC	**Inter-process communication** (IPC) is a set of techniques for the exchange of data among multiple threads in one or more processes. Processes may be running on one or more computers connected by a network. IPC techniques are divided into methods for message passing, synchronization, shared memory, and *remote procedure calls* (RPC). The method of IPC used may vary based on the bandwidth and latency of communication between the threads, and the type of data being communicated. (//wikipedia)
Kernel	The central component of most computer operating systems, bridging between software applications and the data processing carried out by the hardware. The kernel manages centralized resources such as CPU allocation, memory allocation, disk usage, networking access. The kernel usually provides a convenient abstraction so that user programs do not need to be concerned with the actual physical hardware that is being used, but can make use of kernel facilities (a kind of virtual machine) instead. http://en.wikipedia.org/wiki/Kernel_%28computer_science%29
Library (static, shared)	A library is a collection of object files, providing a collection of symbols and machine code (instructions, subroutines and/or processes or *classes*, depending on the language), used to provide functionalities to other programs when creating software. Libraries can be in *object* or *source form*. They tend to be classified between static and shared but can also include remote libraries (according to the way the code is combined into the application program that is using it). ● **Static library:** A static library, also known as an archive, consists of a set of routines or functions of which a subset (all needed routines) are copied into a target application by the compiler, linker, or binder, producing object files and a stand-alone executable file. Common formats: .lib (Microsoft Windows) or .a (POSIX). See *static linking* in main text. ● **Shared library**: A shared library is a file (resident in a computer system and not linked at compile time into an executable) that contains a set of functionalities that can be used (called) by a variety of applications at the same time, at loadtime or runtime. A shared library is thus a collection of object files that have been specifically prepared to be loaded into memory and then executed by other applications (so-called "shared objects"). The typical use of a shared library is to contain common functions that can be executed without being duplicated in every executable that uses that function. The library may be on the disk or in memory (RAM sharing). Common formats: .dll (Microsoft Windows) or .so (POSIX). See ***dynamic linking*** in main text. ● **Remote libraries**: these are shared libraries containing functionalities that are called (requested) using *remote procedure calls* (see ***RPC*** in the main text), typicall over a network. ● **Object / class libraries** // (TBD...) Libraries often “expose” their functions (allow other programs to use their functions) via an *interface* (see definition). **See also:** *Linker* **Comment**: as an analogy in the legal world, static libraries could be considered like standard (boilerplate) clauses to be built into contracts (the contract

	being the application or main program) and dynamic libraries are more like statutory provisions that can be referred to. *// Mention dynamically loadable objects and libraries for languages like Java and Python?*
Linker	A linker is a program tool (or a set of programs) that combines *object files* from multiple sources (including *static libraries* and/or *shared libraries*) to produce an *executable* or, if the executable is part of a larger system, a *library (static* or *shared*). Typically, an application/program is made up of a series of component parts, that are combined (compiled or linked) to create the complete program (an executable or run-time program). Often, these components are commonly-used functions that are stored in other (object) files called *libraries* (see definition). For example, a program may use common functions that many other programs use as well, e.g. common graphics routines, or network communication, that may be stored in either *static* or *shared* libraries. A linker incorporates these external functions into the main program. The output file of a linker for **statically linked objects** has all dependencies between objects resolved at compile time and does not usually contain external references to unknown symbols. This means that the linker *copies* all (or some) of the object files present in the library into the resulting executable. For **dynamically loadable objects** or *shared libraries*, the external references are resolved at runtime (and loaded into RAM). Note that a linker need not resolve all the external references when producing a (static) program, but can resolve some of them at runtime (too). So a linker is used on two separate occasions: (1) as part of the process used to produce a program on disk, and then (again) (2) to take the program from disk and put it in memory. 1. When a linker is used to produce a program on disk, it takes a collection of object files, calculates how the object files interoperate (e.g. which object files use functions defined in other object files) and replaces external references with resolved references. This is commonly called ***static linking*** (see main article). The resulting program combining the functions is written to disk, along with information about which external references remain and where to look for those external references (these external references can be resolved on runtime – see 2.). . 2. Runtime linking involves reading the program from disk into memory, then reading the externally referenced code (e.g. a shared library) from disk into memory too, to create a complete program in memory. This is also commonly called "***run time linking"***, "***loading***" or "***dynamic linking***" - see main article on dynamic linking . See also: Object file, Executable, Shared library **Comment 1**: See example code in Annex 1 (thanks, ade). *// maybe make the difference between this linker and dld (A: for next draft....?)* *// Binder seems to be a form of a Linker in Delphi*
LKM	Loadable Kernel Module (or Linux Kernel Module, for Linux): an object file that contains code to extend the running kernel, or so-called *base kernel*, of an operating system. LKMs are typically used to add support for new hardware and/or filesystems, or for adding system calls. When the functionality provided by an LKM is no longer required, it can be unloaded in order to free memory. LKM can also be considered a form of *plug-in* (see definition and discussion).

	Source: wikipedia, ibm, other http://www.ibm.com/developerworks/linux/library/l-lkm/index.html?ca=dgr-lnxw07LinuxLKM&S_TACT=105AGX59&S_CMP=GR http://www.linux.org/docs/ldp/howto/Module-HOWTO/copyright.html
Machine code (a.k.a. object code)	Machine code is a series of instructions that can be executed directly by the machine hardware. A piece of machine code (represented as a series of bytes) placed in computer memory can be read and executed directly by a machine's instruction units. **Comments**: 1. Object code is also protected by copyright, either as a separate work or a derivative work of the underlying source (the second giving rise to the debate on who is the copyright holder of the object code), or as a different form of the same work, i.e. a simple copy of the source code. 2. The executable files you run on your computer (like NOTEPAD.EXE or /usr/bin/vim) contain machine code. Machine code is highly specific for a particular type of processor. Processors from different manufacturers can typically not execute each others' machine code.
Object file	An object file is a "machine readable" file that contains *machine code* or *byte code* and symbol information. The symbol information represents the dependencies between this object file and other object files and usually contains a list of names of functions and/or variables that: • the machine code in this object file references, but which are not defined in this file. These symbols represent a dependency on another object file that should define these symbols. • are defined in this object file and which could be referenced from other object files (they have been "exported"). **Comment**: An object file is usually not directly executable by the operating system but is linked (by a linker) with other files to form an executable or a *library*. See also: Linker
Pipes	A pipe is a command-line operator available in some operating systems whereby a number of processes (tasks), whose names are listed sequentially, are activated in specified order so that each process (after the first-named) accepts as its input the output from the immediately previously named process. Source: [JOHN DAINTITH. "pipe." A Dictionary of Computing. 2004. *Encyclopedia.com.* 7 Jul. 2009 <http://www.encyclopedia.com>.]
Plug-in	A plug-in is a piece of machine or source code (i.e. a program) that is loaded into a running system or “host” and then executed. Usually the plug-in conforms to an *interface specification* (API) of the host, which allows the host to delegate certain *function calls* to the plug-in which can then execute them in any way it sees fit. This is similar to a *library*, however the plug-in generally extends the functions of a program/system with additional functionality, while a library is often a necessary part of the original functioning of a program or kernel/operating system. Plug-ins come in different flavours. Two major ones are: • *Machine code* plug-ins: Contain (as the name implies) machine code and are loaded *[on the fly?]* using technology akin to *dynamic linking*. Examples of machine code plug-ins are *Linux Kernel Modules* and SANE backend drivers, GIMP plug-ins, Eclipse Extensions. • *Script* plug-ins: Contain instructions that have to be executed by an *interpreter* of some programming language. Examples of script plug-ins are

	FireFox extensions (Javascript) or Perl DBI drivers. Plug-ins can be self-contained or made to use the API of a host so as to communicate (make calls) with the host. **Comment**: Plug-ins are typically used to extend a system with additional functionality or implement a common functionality for a specific type of device. By supporting plug-ins, a program author makes his work extensible by third parties. Usually the host system does not know in advance what plug-in it will load at run-time. See *plug-in* in main text relating to methods for interaction between programs/kernels and plug-ins.
Preprocessor	A preprocessor is a program (or set of programs) that converts a *source file* into an intermediate form of source file by interpreting special preprocessor instructions in the original source file and making changes to the this file (as directed by the preprocessor instructions). Typical preprocessor operations are: • Verbatim inclusion of include files • Substitution of macro calls by their expansion • Conditional inclusion of parts of a source file based on settings in the environment
Procedure / function	A sequence of source code statements or machine code (depending on whether one is considering before- or after-compilation), usually with a name, which realises some specific functionality when executed. Most procedures operate on data held in the memory of the computer where they are executed, or cause only local changes (for instance, procedures to place text on the screen or to read data from disk). Some procedures may be "remote procedures" in the sense that their function in the local machine is merely to invoke (through network data communication) a function in some remote location (e.g. another machine somewhere else); the actual computation and effectuation of functionality is done elsewhere. The results of the procedure which is executed remotely are returned to the local machine for further processing.
Program	While there is no specific legal definition (on an EU-wide basis), for the purposes of this document we use the following definition: *a sequence of statements or instructions expressed in words, codes, schemes or in any other form, which is capable, when incorporated in a machine-readable medium, of causing a computer (a device with information processing capabilities) to perform a task or achieve a particular result.* [WIPO Model provisions on protection of software, 1978/restated 1991] **Background**: Computer programs or "applications" are either (a) developed (written) in *source code*, which is then *compiled* to *object code*, and linked with external *libraries* to create *executables*, which can be executed by the computer system hardware, or else (b) not compiled but their code is *interpreted* at run time and/or *dynamically* linked *to* external libraries to provide certain functionalities or services for the application/program. **Comment**: The GPLv3 defines "Program" (for its purposes) as "*any copyrightable work licensed under this License*". GPLv2 is similar, referring to "*program or other work*". However as this document aims to comment only on software interactions or interoperation, we limit ourselves here to software programs. **Legal note**: Not all programs are copyrightable, particularly if they lack the minimum level of originality, or for other reasons they are excluded from copyright. There is discussion whether an "*interface*" is copyrightable (see definition below), however we note that while the EUCPD excludes the ideas and principles that underlie a computer program's interfaces – it does not necessarily exclude from copyright any code (interface) that implements

	those ideas, or the text that defines the interface (API definitions, conventions, etc.).
Script	A script is a set of instructions (usually *source code* in a *source file*) that are usually interpreted by an interpreter to control the behaviour of one or more software applications or to perform specific tasks. Scripts are typically short programs, such as an installation or configuration script. **Comment**: A boot script is a specific set of instructions that are run at boot time for some specific purpose. See also: *Interpreter*
Socket	A socket is the endpoint of a bidirectional communication flow across a (Internet Protocol-based) computer network, e.g. between applications across a network. Thus *RPCs* use sockets for communicating between applications.
Source code	The human readable text contained in a *source file*, mainly consisting of instructions (code) for a computer and certain comments for developers. Source code is normally considered the preferred form for developing (creating or modifying) a program by a human being. Apart from instructions, source code can also consist in: • Comments intended for a human reader (e.g. the copyright notice is usually placed in the source file using a comment). • Makefiles, autoconf/automake files for building a computer program or parts of it. • Meta-instructions for the compiler, preprocessor or interpreter(for instance giving hints on how to generate/execute certain instructions more efficiently, macro definitions, etc.). Reference: GPLv3 (cl1.1); GPLv2 (cl.3); MPL 1.1 (cl 1.1), OSL 3.0 (cl3). See also: Source file, Compiler , Machine code , Byte code , Preprocessor **Background**: Computer hardware cannot usually execute source code instructions directly and these should be either: • Translated to machine code using a compiler, • Translated to byte code using a compiler and interpreted by an interpreter, or • Interpreted by an interpreter **Relevance**: One needs to understand source code and machine code in order to understand many other concepts (such as compilation and translation), which in turn are relevant for appraising questions such as who performs the act of physically combining two code-sets. And this may be an element in appraising who creates a derivative work, which in turn has implications under licenses such as the GPL. Copyleft licenses such as the GPL, MPL, OSL, etc. include obligations to accompany, offer access to or otherwise make available the “source code” of a computer program that has been distributed in object / executable form. Note that the scope of this source code obligation may vary from license to license. See comment 2 below. **Comments** **1.** Source code is protected by copyright as a literary work, so long as it has a sufficient level of originality. However, there are questions whether certain parts of source code (including variable declarations, identifiers in header files) have copyright protection. **2.** For the purpose of this document, we exclude icons, sound files and other “mere data” (as opposed to “instructions”) from the definition of source code, although these items may well either be included in a program or themselves consist of computer files (vector files, etc.).

	3: GPLv3 includes as part of “Corresponding Source” (and thus implicitly within the definition of Source Code), “a*ll the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities* …. [including] *interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require.* Similarly, the MPL includes in its definition of source code: “... plus any *associated interface definition files, scripts used to control compilation and installation of an Executable, or source code differential comparisons*”. In addition to specifying the source code obligation under these licenses, this also gives an idea of what a licensor (or the community) may, in certain circumstances, consider as “source code” (e.g. where no specific definition of source code is given). *// GPL interpretation of "human readable". Is assembler human readable? Perl? APL? Debian doesn't see generated assembler or even C source code as the real source code. Debian wants the source be delivered in the form which is most convenient for editing (which certainly is not some form of generated code (MK).*
Source file	A source file is a computer file that contains *source code* for a computer.
Symbol	Representation or reference of external functionality contained in another part or component of a program (library, etc.). I.e. name referring to either a specific location in an object file (also called a label, then) so that you can talk about "function usage() in this object file" *or* a name referring to an external symbol (e.g. a name which refers to something "out there" which must be resolved by providing an object file which contains that symbol).
Systems calls / Function calls	**System call**: mechanism used by an application program to request service from the operating system (load, execute, get, put, create, read, write, etc.). **Function call** (to subroutine): mechanism used by a program to execute and request the result of a function or routine (which may be contained in another file / library or part of the system).
Variable	A variable is a symbol, facility or identifier for storing and representing data. Variables can represent numerical values, character strings, memory addresses (e.g. where data is or can be stored), etc. A variable has one (or more) names and a data type (characteristic relating to the type of data that it represents or stores).
Virtual machine	A virtual machine is a program (or set of programs) that emulates another (existing or idealized) machine environment. Some virtual machines emulate an existing CPU type and machine, and can be used to simulate a computer plus operating system as an application on another operating system (e.g. VMWare). Other virtual machines emulate an idealized machine and are used to run programs that have been compiled to the byte code of that virtual machine (e.g. a Java Virtual Machine).

Legal Definitions	
Adaptation / Transformation	See Derivative work. These are actions that normally create derivative works. **Comment** UK: *(a) in relation to a literary or dramatic work, [adaptation] means—* *(i) a translation of the work; .... In relation to a computer program a "translation" includes a version of the program in which it is converted into or out of a computer language or code or into a different computer language or code, otherwise than incidentally in the course of running the program.* **Relevance**: wrt to object code... (?) - compiled object code seen a adaptation of the original source code (and thus not a mere reproduction).
Aggregate (work)	An **aggregate (work)** is a term used by the GPL that is not defined by copyright laws, but is understood to be a collection of independent programs or works which is not a collective work or compilation, i.e. when there is no new work of authorship or intellectual creation created (as meant by the Berne Convention). A person putting a bunch of independent software programs on a CD or web-server, e.g. for sharing with friends or downloading by the public, would be creating an aggregate of different works.
Collective work / Composed work or compilation	In this document **collective work** means collections of literary works (including computer programs) such as encyclopedias and anthologies which, by reason of the selection and arrangement of their contents, constitute intellectual creations. [Based on Berne Convention, Article 2 (5) and WIPO Copyright Treaty, Article 4]. A **composed work** (or "**compilation**") is the collection or assembly of preexisting works so as to create a new work of authorship (and thus may include "collective" works), usually without the collaboration or "active contribution" by the authors of the component parts. So, a person creating a new program by combining his/her own new work and several third party libraries (or creating a driver plug in and distributing it loaded together with a third party operating system) is probably creating a composed work or compilation. **Relevance**. Most computer programs are made up of a series of different component parts. The concept of a composed work/compilation is particularly relevant for most free software development, as nearly all programs today include third party components, libraries, interfaces, plug-ins, etc. and interact or interoperate with other programs in (legally) meaningful ways. This document aims to discuss the legal consequences of creating such collective works (as well as derivative works). **Comments:** **1**: The ownership of rights in the new work as a whole (the collective or composed work) is separate from the ownership of rights in the contributed parts or subcomponents. Usually, the person who edited (supervised, organised, selected and arranged) the collective work is owner of copyright in this new work. Except if rights have been assigned, each contributor remains copyright holder of his/her contribution. **2**: In certain jurisdictions, whether a "collective" work or composed work is created depends on how the collecting or arranging is done and who does it. In continental EU jurisdictions, a distinguishing factor in the concept of "collective work" (in contrast with a mere *compilation* or *composed work)*, is the fact that the component parts are commissioned or intentionally "contributed" to the whole (i.e. with a degree of intent or commission on the part of the editor and authors to create the collective work. (// *[confirm in jurisdictions]).*

	3. However, it is often difficult to distinguish what is a collective, compilation, derivative, or even joint, work, particularly within the free software development model where there are different forms of project governance and organisation, with certain modules being "contributed" within a roadmap, others being freely "used" (third party libraries) without the express contribution (or intention to contribute) of the rights holder of those libraries. **4:** To create a collective work or compilation, the creator must be authorised by the owners of the copyright on the component parts to use those parts in a certain manner. When the resulting work is redistributed, there is a debate (see below) whether this authorisation should consist of merely reproducing (and subsequently distributing) the component, or also modifying/transforming and distributing it. As regards the GPL, whose terms as to reproducing/distributing and modifying/distributing are different, the question is important. **5:** There is significant debate (to which this document strives to contribute) as to whether a composed work or compilation is either (a) also a derivative work of the component parts or (b) otherwise covered by the GPL provisions as to "works based on the Program" - something which may depend on the methods of combining these with the new program (see the main articles on the different forms of interoperation/interaction). - If the component parts are themselves modified so as to be integrated in the new program, the author is at least creating a derived work of the original component. But the code he/she has written, to integrate or use those components in the new program, is not necessarily a derivative work of those components. - if the components are statically or dynamically linked into the new program, "plugged in" or called up, see main article on these forms of interaction.
Copyleft	**Copyleft** refers to a specific type of provision contained in certain free software licenses. This term stipulates that in order for the licensee to be entitled to redistribute the program and/or any eventual modifications to (and potentially "works based on") the program, he/she must do so under the terms of the same license. If the distribution is in binary or non-source form, he/she must also provide some means of access to the source code. **Comments:** Generally speaking (and also "debatably"): 1. If the scope of the copyleft obligations is limited to the original work and/or direct modifications of and/or additions to the original work, often on a file level, the original work, or parts thereof, then the license is considered to be a **weak copyleft license**. Examples of this type of license include the Mozilla Public License (MPL). Similar "weak" copyleft licenses, but with a somewhat wider copyleft obligation (applicable to "derivative works as defined by copyright law), are the Open Software License (OSL) or the Common Public License (CPL). 2. If the scope of the copyleft obligations is aimed to cover all derivatives of the original work (through whatever means, as discussed in this paper), and arguably other works which, while not being "derivative works" (or arguably so), are based on the original work and are combined with the original work to create a larger work, then the license is considered a **strong copyleft license**. An example of this type of license is the GNU General Public License, versions 2 and 3 (GPLv2, GPLv3). The extent of the scope of strong copyleft license obligations is subject of much discussion and the purpose of this document is to facilitate such discussion in relation to software interactions.
Derivative work	The definition and scope of a **derivative work** (and *collective/compiled works*, see below) is a subject of much debate, complicated by differences between jurisdictions. Due to this we have resorted to a very general definition, based on the Berne Convention and WIPO Copyright Treaty. In addition, we present some general discussion on derivative works in different jurisdictions.

In this document, **derivative work** means *translations, adaptations, arrangements and other alterations* of a literary work, such as a computer program [Based on Berne Convention, Art. 2(3) and WIPO Copyright Treaty, Art. 4; and EUCPD Art. 4(3)]

> *Art. 2.3 Berne: Translations, adaptations, arrangements of music and other alterations of a literary or artistic work shall be protected as original works without prejudice to the copyright in the original work*
> *Art. 4(3) EUCPD: translation, adaptation, arrangement and any other alteration of a computer program and the reproduction of the results thereof.*

Thus, generally speaking, derivative works in respect of programs could be created by:

1. a (straightforward) *alteration* of the existing work – e.g. by modifying the lines of code; or
2. a *combination* of new work with the existing work – e.g. by adding new code to existing code; or
3. other *interoperation* between separate works, if either or both of the works are dependent of the other to such extent that a court considers the whole to be derivative of either (or both). This is where the main lines of debate lie.

(this list is not exhaustive)

However, appraisal of whether a work is derivative or independent is traditionally a case-by-case decision and few, if any, court decisions exist within the FOSS environment.

While the alteration of an existing work as derivative work is not controversial (changing lines of code, adding new lines in a file), the effect of combining works or creating a significant form of interoperation between them gives rise to a debate (for which this document strives to provide some input).

Relevance: Under copyright laws, a derivative work may not be created or distributed without permission of the copyright holder of the original work, which is granted in a license. If the license terms of the original work pose conditions on the creation or subsequent distribution of any derivative work or on exploitation of the original work as a part of a derivative work (such as in GPLv2 or GPLv3), then the question whether a work is a derivative work of another is highly relevant.

Comments:

1. Jurisdiction: Copyright is a national right and therefore national laws need to be consulted in order to understand how that national copyright law treats the question of derivative works (and also interpret the wording of the license in question). e.g. in England, there is no codified definition of a derivative work and essentially you need to look at whether the work in question infringes the other. E.g. Infringement could be found where there is copy of the whole or a substantial part of the work in the new work and also rests on the question of the effort and the exercise of choice using skill and judgment.

2. Jurisdiction (USA): While there is substantial US case law on GUI copyright and related areas, copyright litigation and reported cases at the machine operating level is limited. However, certain US courts (9th Circuit) have considered the question of whether one work is a derivative of the other by looking at:

(a) the expressed ideas within the programs (abstraction);

(b) removing the elements which are not capable of copyright protection (filtration); and

(c) comparing the resulting two works (comparison).

	If the outcome is “substantially similar” then the new work is a copyright infringement of the other. ***3.*** *Jurisdiction (USA bis):* the US concept of a derivative work being “based on” a pre-existing work – which seems much wider than “translation, adaptation or other alteration” - is not found in EU legislation (though the concept may be taken into account in national jurisprudence). // This may be balanced by a wider application of the concept of “fair use” of a work. **4.** Compiling: there seems to be a debate on whether a compiling code results in a copy of the original work or a transformation (modification). Again, this may be jurisdiction specific and, in respect of the GPL, would lead to different obligations arising on redistribution of the code.
Distribution	**Distribution** involves the making available, issuing to the public or putting into circulation (of) the original and copies of the work (including rental). This is limited to transferring (a copy of) the work in tangible form (CD, etc.). So called “distribution” of works in intangible form over a network is usually covered by the concept of “public communication” (see definition).
Joint work	**Joint work** means a single work which has been created by two or more authors in a way that the part created by one is not an independent work. Two developers working together on the same file at the same time, or commenting and building on each others work (extreme programming, etc.) would be creating a joint work. In most countries, such authors hold the copyright to the work jointly. There are differences as to how joint copyright can be exercised, either independently or with the consent of all joint owners. In some countries licensing requires joint decision, in others (US) each joint author can exploit the whole, whereas enforcing can be done by each author separately. However, in other countries also enforcing requires joint decision or participation in legal proceedings. **Relevance**: This concept is more relevant to ownership of rights rather than software interactions, as it requires the active participation of two or more authors to create a single work, and not necessarily the independent use of third party code. **Comments** **1:** Generally speaking, to create a unitary work of joint authorship, the contributions must be combined inseparably into a “whole” (interdependent or inseparable) – otherwise we are looking at a collective or compiled work (with independent ownership of the parts but, depending on who does the arrangement, potentially joint owners of the whole as a collective or composed work). **2.** In the USA, it seems the question of intent is important: joint owners intend their works to be merged into inseparable or interdependent parts of a whole. This is often not the case in free software development (except for close-knit projects such as Linux kernel or Mozilla core, where developers are designated or volunteer to make specific components to be merged into the core/kernel).
Public communication	**Public communication** includes “*making available to the public of a work in such a way that members of the public may access the work from a place and at a time individually chosen by them (by wire or wireless means)*”. i.e. it covers so-called online “distribution”. However, the EUCPD does not itself refer to public communication of computer programs. Depending on the member state, the sharing or transmission of works in intangible form over a network could fall either under the legal concept of “distribution”, or the general rules for all works will apply to software subsidiarily and this action will be covered by the “public communication”). In both cases, the consent of the rightsholder is required. NB: in this paper, for ease of understanding, we have used the term “Distribution” to cover both tangible and intangible transfer of software from one person or server to another.
Reproduction (copying)	Permanent or temporary **reproduction** (copying) of a computer program by any means and in any form, in part or in whole. **Comment**: Insofar as loading, displaying, running, transmission or storage of the computer program necessitate such reproduction, such acts shall be

	subject to authorization by the rightholder. However, in some countries there are provisions under which such permission is granted by law, unless it has been reserved in a license or otherwise.
“Use”	No legal definition, though the ECPD and national laws refer to “legitimate users”. It is of course understood that to use a program you must make a reproduction, both when downloading and installing it, and also in memory when executing it.
Work	The result of the **work** of authorship of a human being (or a machine on the instructions of a human being). Thus – for our purposes - “work” includes written text, graphics, diagrams, source code (a form of written text), object code, etc. **Comments**: **1.** To obtain copyright protection, the work must have a minimum degree of originality (the required level of which can vary from jurisdiction to jurisdiction). **2.** Machine created works (including, arguably, object code which is compiled by a compiler) are generally considered the work of the person who arranged for the machine to create the result. **3.** Copyright protection does not extend to the principles and ideas which underlie a work (TRIPS and EUCD) (in the US, the full exclusion reads: *any idea, procedure, process, system, method of operation, concept, principle, or discovery, regardless of the form in which it is described, explained, illustrated, or embodied in such work).* The consequence of this may be that the invoking of the functions of a program (a process/ a whole system) may not be something that the rightsholder of the code embodying those functions can control (via copyright – though contractual obligations could apply).