LIABILITY ISSUES

WHO IS LIABLE FOR SOFTWARE ERRORS? PROPOSED NEW PRODUCT LIABILITY LAW IN AUSTRALIA

Fixing liability for the consequences of defective software is a very difficult matter for the law to deal with. Software has many functions and applications and is frequently dependent upon the operation of other technology to discharge its task correctly and efficiently. What steps can a legislature take in introducing product liability legislation for software to ensure that the legal terminology accurately defines the technical situation so as to categorise software correctly and attach liability accordingly. Roger Clarke develops these ideas reporting on a proposed new product liability law in Australia. Michael Scott then examines some liability issues in medicine with regard to patient care systems.

When a person suffers as a result of *'computer error'*, does the injured party have the ability to seek redress from the person or organisation responsible?

Simple though the question may seem, there is no straightforward answer to it. Depending on the circumstances, there are several heads of law under which an action might be initiated. The most common of these are contract law (which applies only to the parties to a contract) and the tort of negligence (under which everyone has a limited duty of care to everyone else).

Another possibility is *'product liability law'*, which imposes some responsibilities on the seller and/or the original manufacturer of a product. The Commonwealth Attorney-General has requested the Australian Law Reform Commission (ALRC) to study the state of product liability law in Australia, and recommend changes. Some key aspects of the Commission's proposals are outlined in Exhibit 1.

EXHIBIT 1: THE PROPOSED NEW PRODUCT LIABILITY LAW

1. The Australian Law Reform Commission proposed that, as a general principle, losses caused by products should be reflected in the price of the goods, to the extent that those losses are a consequence of the activity of a person in the chain of production. A new regime is needed to give effect to that principle probably by passing a new Commonwealth statute.
2. Liability is to be imposed on the *'production enterprise'* (the chain of production) for *'loss or damage'* caused by *'goods'*, provided that those goods can be shown to have been built in an *'unsafe'* (or perhaps an 'unacceptable') condition, and that that condition *'caused'* the loss or damage.
3. The *'production enterprise'* consists of all persons *"engaged in the process of producing the goods and putting them into commercial circulation"*. Since the *'production enterprise'* is generally not a legal person, a *'primary defendant'* is defined, who the affected party may sue. In general this is the retailer.
4. *'Goods'* are distinguished from *'services'*, but no clear definition is offered. Discussion is provided concerning the meanings of *'causation'*, *'unsafe'* and *'unacceptable'*. The onus is on the *'primary defendant'* to show that the condition causing the harm was not in existence at the time the goods left the production enterprise, and/or that, despite the existence of that condition, the goods were safe.
5. The *'primary defendant'* has the right to recover from other participants in the *'production enterprise'* and from persons outside the enterprise whose activities caused or contributed to the harm.
6. *'Loss or damage'* of the following kinds is to be compensated:
   - personal injury;
   - property damage;
   - consequential economic and non-economic loss (i.e. arising from personal injury or property damage); and
   - *'pure'* economic loss (i.e. which occurs without any injury or damage), which is suffered by the owner of the goods

Not to be compensated are:
- pure economic loss suffered by third parties; and
- pure non-economic loss suffered by any party.

Reference: Law Reform Commission (Australia) Discussion Paper No.34: 'Product Liability' August 1988

The existing and any amended law would undoubtedly apply to computer hardware. But is computer software a product for the purposes of product liability law? The current law is sufficiently unclear that different lawyers might well give different answers, for the very good reason that, if someone felt moved to finance a test case, different judges might well give different judgements

The term *'goods'* is used in the existing *Trade Practices Act 1974*, but is not defined. The intention of the ALRC's new proposals is that it apply to a complete product (i.e. including subsidiary components) supplies from a *'production enterprise'* to some party who was, at least potentially, a user of the product. It would exclude services, but would include products supplied under a contract for services.

This article assesses whether, under the new law the ALRC proposes, any circumstances would exist under which software would be goods, and therefore be subject to product liability law. It is useful to look firstly at the rather simpler question of whether data can be *'goods'*.

LIABILITY FOR DAMAGE CAUSED BY DATA

Data may be sold as a product. A book is a good, but the text within it, and the words and letters that make up the text, are neither a good, nor even a component of the *'whole product'*. The rationale for this appears to be that data is inert, and cannot play a role in the function that the good performs. In the same manner, it would appear that both the physical device (such as a disk-drive or cassette-player) and the physical medium on which data is delivered (such as magnetic disk or cassette, or CD-ROM) are goods. Therefore harm arising from a defect in the equipment or medium would be subject to product liability law. However, data stored on such media is not deemed to be a component of the *'complete product'*. Data such as a dictionary, encyclopaedia or mailing list might therefore be thought of by seller and buyer as a product, but it would not be subject to product liability law.

The owners of public access databases (such as the I.P. Sharp collection of economic data housed in Toronto, or the

Australian legal database CLIRS) would therefore not be subject to product liability law where the data is distributed on optical disks. (If those databases are accessed remotely by terminal or PC, their owners are also not liable, in this case because data access is of the nature of a service, not a good). So, in general, it appears that data is not subject to product liability law. This has potential implications for software, which are discussed later.

'SOFTWARE' AS GOODS

The term *'software'* is used ambiguously. At its most abstract, it refers to a set of instructions intended to cause a computing device to perform particular functions. However, it is also used in a more restrictive (and original) sense, to refer to only such sets of instructions as are stored externally to, and independently of, the machine they are used in.

In order to appreciate whether software would be subject to the new product liability law, it is necessary to consider firstly software which is intrinsic to a computer, and then software which is loaded into a computer from an external medium.

EXHIBIT 2: CLASSES OF SOFTWARE

- intrinsic software
  - embedded in hardware (hardware or VLSI)
  - embedded in firmware (ROMS)
    - inserted in the computer at time of purchase
    - inserted in the computer later, as optional extras
- extrinsic software
  (stored on and loaded from an external medium)

LIABILITY FOR HARM ARISING FROM INTRINSIC SOFTWARE

By *'intrinsic software'*, I mean software which can be easily argued to be a component of a *'complete product'*. There are several different classes of *'intrinsic software'*.

The first of these is software which is embedded in hardware. Early models of computers in the 1940's and 1950's featured *'hard-wired'* programs, and the characteristic has reappeared in the 1980's in the form of VLSI (Very Large Scale Integration). In effect, the potentially general-purpose computer has been wired in the factory to perform only certain very specific functions. This is common in appliances such as washing machines and ovens, and in electronic ignitions in cars.

The second class is software which is embedded in firmware. This refers to a popular kind of computer architecture in which a general-purpose computer is provided with special-purpose capabilities by including pre-programmed ROMs (Read-Only Memory modules). The result is that the product delivered from the computer factory has certain functions built-in, but is still capable of being used for a wide variety of purposes, simply by loading further programs from external media like cassettes or disks. A common example is the BASIC interpreter which is embedded in the ROMs of many micros of the late 1970's and 1980's.

The third class of intrinsic software is programs which are embedded in optional-extra ROMs, i.e. Read-Only Memory modules which may be purchased separately from a computer, and added to it later. Such optional-extra ROMs are commonly referred to as *'add-on boards'* in the IBM PC arena, and have been widely used to achieve follow-on sales in the hobby-computer market (Ataris and Commodores).

Goods containing intrinsic software appear to be generally subject to product liability law. In the first two cases, the good is the complete computer, including software, as delivered by the retailer. If the retailer assembled the product, he may have to carry the majority of the liability, but if the product was essentially complete when it left the factory, then the main risk would be borne by the manufacturer.

The Commission's proposals appear to achieve their intention of ensuring that harm arising from unsafe software is paid for by the *'production enterprise'*, i.e. the IT industry.

In the third case, the computer is one good and the optional-extra ROM is another. Even here, the Commission's proposals appear on the surface to achieve their objectives, although the situation is more complex, and this may provide retailers with some additional scope for avoiding liability.

Disputes often arise between the suppliers of the various elements of a complete system, with each being able to demonstrate that their own product independently of the others, performs according to specifications (and therefore, in the Commission's terms, *'safely'* and *'acceptably'*). In many such cases it is economically impracticable (sometimes perhaps even technically impossible) to prove which supplier is at fault.

It is unclear how the Commission's proposals would overcome such an impasse. Both potential *'primary defendants'* (in general the retailers of the computer and of the separately purchased software) may successfully show that there was no characteristic of either good which caused the loss or damage, and the aggrieved party may be unable to find anyone from whom he can gain compensation. (Although it provides little comfort, this weakness in the Commission's proposals would appear to apply more generally than only to software).

LIABILITY FOR HARM ARISING FROM EXTRINSIC SOFTWARE

There are many instances in which software is genuinely *'soft'*, in the sense of not being part of the machine, but instead being located from an external medium into the computer's high-speed main memory (in recent years often called RAM – Random Access Memory).

To date, economic factors have almost always dictated that main memory is limited in size and ephemeral rather than persistent (i.e. its contents are lost when electric power is removed). It is therefore generally necessary to re-load the software from the external medium on each occasion it is required.

It is clear that the external medium on which such software is stored is a good, and product liability would apply to it. However, the software does not appear to be a good in its own right, because while it is on the medium, it is as inert as data on a diskette or text in a book. The combination of medium-and-contents does not appear to be a *'complete product'*, because the contents do not enable the medium to do anything – they enable an entirely different good (a computer) to perform some particular function.

If this line of reasoning is right, then extrinsic software may not be subject to product liability law, even if it is unsafe or unacceptable. So a software manufacturer would be exposed to public liability risks if the software is intrinsic, but not if it is extrinsic. Hence, depending on the medium on which the software is delivered by the retailer, he may not be liable. This seems anomalous, particularly if the software manufacturer has no control over the medium on which the software is delivered to the eventual consumer.

If the ALRC wishes to deal comprehensively with computer software, then a number of further factors need to be considered (see Exhibit 3).

EXHIBIT 3: OTHER POTENTIALLY RELEVANT FACTORS
- purchase with the computer vs. separate purchase
- pre-packaged vs. custombuilt software
- active (real-time) vs. human-mediated systems
- systems vs. utility vs. application software
- software form

One is whether the software is purchased with the hardware or separately from it. Where the software is purchased with the hardware, the software might be argued to be a component of the *'complete product'*. If that argument were successful, then it would be subject to product liability law in the same way as intrinsic software. Where software is purchased separately, such an argument is far more difficult. If this distinction were intentionally made, or arose from case law, it would create an incentive for suppliers to contrive to deliver software separately from hardware, in order to avoid the risk of public liability.

Another factor is whether the software is pre-packaged or custom-built. Pre-packaged software might be argued to be a product, and therefore to have at least some of the characteristics of a good (although perhaps not enough of them for the courts to treat it as such). With custom-built software, it is much easier to argue instead that the software is of the nature of a service. (In the not infrequent case of custom-built software being subsequently packaged for sale to further clients, the *'productisation'* of the software would presumably not affect the status of the first installation). A third factor is whether the software directly causes physical action, or its output is mediated by a human. In active systems (e.g. real-time control of chemical processes and environments, and navigation systems), decision-making on matters of real consequence is delegated to an artefact. Since the scope for harm may be substantial, it might be particularly desirable for the risk to be borne by the *'production enterprise'*, and explicitly costed into the product(s). In passive systems some person uses the output, and existing laws, particularly negligence, may be sufficient to ensure that the software manufacturer has an interest in product quality.

It is conventional to distinguish between *'system software'* and *'application software'*. System software is concerned with the operation of the machine, while application software performs specific functions directly understandable to and desired by the user. A third category, which might be termed *'utility software'*, is emerging, to contain products which have some characteristics of both. In the past, system software was generally purchased from the equipment supplier, and application software more commonly from a third party, but the patterns of supply are now far more varied. I do not believe that these classifications are of much assistance in the area of product liability law.

SOFTWARE FORM

Another very important factor is the rich variety of different forms in which software can exist. Exhibit 4 provides a classification scheme.

EXHIBIT 4: SOFTWARE FORMS

Form of the Source-Code
- expressed as instructions (imperative mood)
- directly executable machine-language
- coded machine-language (hex or octal)
- assembly language
- algorithmic or procedural language (or '3GL')
- expressed as data (descriptive mode)
- problem definition or requirement (e.g. '4GL')
- problem-domain definition (e.g. production-rules)
- empirical knowledge (e.g. in connectionist networks)

Type of Code Translation(s)
- isomorphic, 1-to-1 or 1-to-many (e.g. assemblers and macro-assemblers) vs. non-isomorphic (e.g. compilers)
- one-time, usually in advance (e.g. assemblers, compilers) vs. execution-time (e.g. interpreters)

Form of Object Code
- expressed as directly executable instructions
- expressed as data which require execution-time translation
- rules, needing an expert systems inference engine
- fully interpreted source-code, needing an interpreter
- parameter tables, needing a run-time table processor
- pcode, needing a run-time interpreter

At the point at which software is used, it may exist in directly-executable form (i.e. machine-language, a succession of groups of binary-valued variables which can be successively loaded into a processor's instruction register). However, there are other forms in which software may exist immediately prior to its use. These will be discussed shortly.

Only a tiny proportion of software is created in machine-code. It is usually expressed by a human programmer in some other language, in what is generally referred to as the program's *'source-code'*. Conventional languages comprise a series of commands, expressed in the imperative mood, as instructions for a dumb clerk. Some of these languages are very close to machine-language (e.g. hex, assembler and macro-assembler), while others are much closer to patterns of formal human communications (*'3rd generation'* algorithmic or procedural languages).

However, some languages support moods other than the imperative, and may even preclude imperative expressions. Some of these languages are intended to allow a programmer to describe the characteristics of entities and the relationships between them (schema languages), and some to describe the requirements of the program (sometimes unhelpfully called *'non-procedural'* or *'4th generation' languages)*. Others are concerned with still more abstract issues, such as the description of a whole problem-domain, rather than merely a specific problem (such as expert systems shells), or the capture of raw, empirical knowledge, from which a description of a problem domain might be derived (this is the realm of the emerging neural or connectionist networks).

In order to be used, source-code in any language other than machine-code must be translated. This is generally performed by a piece of software written especially for the purpose. The output of the translation process is referred to as *'object-code'*. In most cases there is only a single step in the translation process, although there are circumstances in which a succession of translations through intermediate languages is advantageous.

(It should be noted that some specialist dictionaries (e.g. *Penguin Dictionary of Computers,* 1985) use software in a very restrictive sense, whereby it must comprise instructions. This would exclude both source-code and object-code that are expressed in the more abstract languages. It is more useful to apply the term *'software'* generically, to refer to all programs whatever their form, provided that they are capable, in practice, of causing a computer to perform a specified or specifiable function. **(See further, the glossary of terms, to be found in Appendix I.)**

The question which remains to be addressed is whether different software forms might be treated differently for the purposes of product liability law.

SOFTWARE AS DATA

Many kinds of software contain data, e.g. some payroll systems contain tax rates and tax thresholds. In the class of software popularly referred to as *'expert systems'* (more precisely, software based on production-rules), data is not merely incidental, but plays a much more central role.

The rules which make up expert systems software may be quite reasonably depicted as data, which needs to be interpreted (in conjunction with additional data provided at run-time) by a particular kind of general-purpose program commonly called an *'inference engine'*.

Expert Systems are not the only kind of software which exhibits data-like characteristics. A great deal of software is delivered to the target computer in the form of directly executable instructions. However, it is quite feasible for software to be delivered as source-code, which the user organisation must translate into directly executable code, which is then stored for later use. The most problematical case is where software is delivered in a form which requires translation every time it is used (see Exhibit 4).

One of the major variants of such translators in the *'interpreter'* for a *'fully interpreted language'* (such as most BASICs and interactive SQLs, as well as expert systems inference engines). Such an interpreter performs substantial translation functions in order to generate directly executable instructions and pass them to the processor.

Another kind of execution-time translator is a *'run-time table processor'*. This uses parameters supplied by the programmer to customise prepared skeletons or templates, and so pass directly executable instructions to the processor. This approach is used in some so-called application generators and 4GLs.

A third variant is a *'run-time interpreter'* which performs far simpler translation of instructions expressed in *'machine-code-like'* instructions (commonly called pcode or pseudo-code).

The case could be easily argued before a court that software delivered in a form which requires the operation of a run-time translator does not comprise instructions, but merely inert data. Since data are not subject to product liability law, such software would also not be subject to product liability law.

Clearly, if software delivered in directly-executable form were to be subject to product liability law, and software delivered in a form requiring run-time translation were not, an incentive would be created to deliver in the latter form. It is already fairly common for **extrinsic** software to be delivered in a form requiring run-time translation. This may well become the norm, at least for application software, as processor power ceases to be a significant constraint, and software manufacturers strive to increase their potential market by delivering portable products.

To date, it has been less common for **intrinsic** software to be delivered in a form requiring run-time translation. However, if there were an incentive to deliver software in that form, then it would not cost manufacturers very much to change their product delivery strategy.

SO CAN WE KNOW WHO IS LIABLE FOR SOFTWARE ERRORS?

There are arguments both for and against software being made subject to product liability law. The software industry would be likely to prefer not to have to carry the risk and pay the insurance costs that go with it. Consumers (both corporate and human) might be expected to prefer the reverse. The worst possible alternative is for the law to remain unclear. If this occurs, consumers and suppliers are forced into expensive litigation, a process in which all sides lose except the lawyers.

It is possible that, depending on the precise wording chosen by legislative draftsmen, and on the interpretations of language imposed by the courts, various forms of software may be deemed to be, or not to be, subject to product liability law. The main sources of difficulty would appear to be those shown in Exhibit 5.

EXHIBIT 5: MAJOR DIFFICULTIES

Circumstances	Reason
● intrinsic software which is supplied by someone other than the hardware supplier	determination of responsibility for a problem which arises from neither in isolation, but only from both together
● extrinsic software	when supplied, the software is inert data, and therefore not a good under product liability law
● software delivered in a form which is not directly executable, but requires run-time translation	when supplied, the software is inert data, and therefore not a good under product liability law

If the analysis in this paper is correct, then the only software which would be subject to product liability law would be intrinsic software supplied with the hardware in directly-executable form.

The process of establishing whether, in law, any such analysis is correct, is fraught with danger and expense. This author's case study of the understanding of information technology shown by Australian courts (*'The Case of the Wombat ROMs'*, Comput. J., 31,1 February 1988) suggests that such basic terms as *'translation'*, *'language'* and *'instruction'* are capable of a wide variety of interpretations. It remains to be seen what confusions such terms as *'run-time interpreter'*, *'pseudo-code'*, *'production-rule'*, *'inference engine'* and *'neural network'* may excite.

The analysis undertaken in this paper suggests that it may be very difficult for a law reform body, and much more so a parliament, to appreciate how to create moderately clear laws relating to software product liability. Moreover, any such scheme might be easily frustrated by software manufacturers.

Roger Clarke
Reader in Information Systems and Head of the Department of Commerce at the Australian National University. He previously spent 17 years in information systems practice, management and consulting. He is Chairman of the Australian Computer Society's Economic, Legal and Social Implications Committee.

APPENDIX I

A WORKING GLOSSARY

It is recommended that readers not familiar with the terms used in this paper consult reference works (such as the *Penguin Dictionary of Computers 1985*) and standard texts. The following informal glossary is intended only to assist in interpretation, and steers well away from complexities, and from problems such as ambiguous and inconsistent usages. In particular, the term *'language'* is used with some hesitancy, because the courts in any particular jurisdiction may decline to recognise a machine-language and/or a programming language as a language for the purposes of that court, e.g. in the interpretation of a copyright statute.

- **data** are measurements, signals or symbols which represent, describe or record some real world phenomenon. Information is data which is relevant to a decision-maker in the context of a particular decision
- **hardware** is the collective word for computers and their ancillary or peripheral equipment. The central and defining element of a computer is the processor. Input devices enable data to be communicated to the processor, and output devices enable processed data to be communicated to human (and other) users
- **the processor** must have *primary storage* available to it. Primary storage is of two types – that which can be read and written whenever the program needs to do so (commonly called *random access memory* – RAM) and that which has had data pre-recorded in it in an unchangeable form (*read-only memory – ROM*). Primary storage is fast, but highly expensive and ephemeral, and so *secondary storage* devices are used to store large quantities of data for long periods. A storage device writes data to and reads data from a *storage medium*. Common secondary storage media include magnetic disks and cassettes, and optical (laser) disks.
- **software** is a collective word for computer programs. A computer program is a set of instructions written with the intention of causing a computer to perform a precisely defined procedure or function.
- in order actually to cause a particular machine to perform that procedure or function, software must be expressed in a particular form called **machine-language** (sometimes referred to as executable or binary code, and sometimes – misleadingly – as object code). Machine-language is peculiar to, and hard-wired into, each family of machines, and comprises a set of primitive terms called its machine-instruction set
- software may be written directly in machine-language, but it is generally more convenient and productive *not* to do so. Instead, most software is written in any of a variety of programming languages. These may be very similar in form to machine-language (assembler or macro-assembler languages), or designed for the convenience of the developer rather than the machine (in particular, algorithmic or procedural languages). The former require the developer laboriously to convert a previously designed problem-solution, whereas the latter support programming in a form very close to the language in which the problem-solver originally expressed the problem-solution (e.g. the Fortran language is suitable for problem-solutions involving formulae).
  In addition to the long-standing assembler and procedural languages, a number of others have been designed to enable the developer to operate at a higher level of abstraction than problem solutions. Some enable the developer to focus not on the problem-solution, but on the problem to be solved or function to be performed. The solution or procedure is delegated to the machine. These are often called fourth-generation or non-procedural languages.
  Others enable the developer to focus not on the problem, but on the micro-world in which a class problem arise (generally called a *'problem-domain'*), e.g. a country's Immigration Act may be encapsulated using such a language, and the resulting software can then be consulted about many different matters regulated by that statute. Not only the problem-solution, but also the definition of the problem, is delegated to the machine. Among this class are logic programming and declarative languages. A development tool of this kind which requires less formal computing science training is **expert systems shells**, which enable problem-domain descriptions to be provided as sets of rules or decision-trees. At this stage in the development of Information Technology, many such products are prototypes rather than well-established products.
  At a still more general (perhaps the ultimate) level of abstraction, languages are emerging from the research laboratories which will enable developers to, at least in some circumstances, merely provide the machine with empirical experience (say a set of cases and their outcomes) and delegate even the definition of the problem-domain on the machine. Research in this area goes under the heading of **rule induction, connectionism and neural networking**
- where software is written in a language other than machine-language, it cannot cause a machine to perform the procedure or function until it has undergone *translation* into machine-language. There may be several translation steps via intermediate languages. The resulting machine-language may then need to undergo a further process called *link-editing*, during which standard library modules are combined with it.
- Several different terms are used for translation, including *assembly* (from a low-level assembler language) and *compilation* (from a high-level procedural language). The input to a translator is called source code, and the output is called object code. In most cases the entire program is translated in advance of its being used, but in some cases the translation is undertaken instruction-by-instruction immediately before the resulting code is executed. In this case the translator is called an *interpreter*